NOKIA 12 GSM MODULE

JAVA™ IMLET PROGRAMMING GUIDE

NOKIA

 Issue 1.0 9231715

Contents

Legal Notice

ACRONYMS AND TERMS

Acronym/term	Description
AM	Application Module A runtime environment for the customer remote application that is connected to the Nokia 12 GSM module via local connectivity. Typically it is a part of the remote device (such as a vending machine) or a separate bridging element.
ANC	Active Naming Context
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CHAP	Challenge Handshake Authentication Protocol
CLDC	Connected Limited Device Configuration
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CSD	Circuit Switched Data
DIS	Device Information Storage
DNS	Domain Name Server
EGPRS	Enhanced General Packet Radio Service
GC	Garbage Collection
GIOP	General Inter-ORB Protocol
GPRS	General Packet Radio Service
GPS	Global Positioning System
GSM	Global System for Mobile Communications
HLA	Home Location Agent
HSCSD	High Speed Circuit Switched Data
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IMEI	International Mobile Equipment Identity
IMlet	Java™ application that runs according to the Information Module Profile (JSR-195)
IMP	Information Module Profile

Acronym/term	Description
I/O	Input/Output
IOR	Interoperable Object Reference
IP	Internet Protocol
JAD	Java Application Descriptor
JAR	Java Archive
J2ME™	Java 2 Micro Edition
J2SE™	Java 2 Standard Edition
MIDP	Mobile Information Device Profile
MIOR	Mobile Interoperable Object Reference
M2M	Machine-to-Machine
MO	Mobile-originated
NITZ	Network Indication and Time Zone
NMEA	National Marine Electronics Association
OMG	Object Management Group
ORB	Object Request Broker
OTA	Over-the-Air
PIN	Personal Identification Number
POA	Portable Object Adapter
PPP	Point-to-Point Protocol
RMS	Record Management System
RTC	Real Time Clock
SDK	Software Development Kit
SIM	Subscriber Indentity Module
SMS	Short Message Service
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UI	User Interface
VM	Virtual Machine
WAP	Wireless Application Protocol
WTK	Wireless Toolkit
WUS	Wake-up Service

1. ABOUT THIS DOCUMENT

This document introduces the reader to Java™ IMlet programming for Nokia 12 GSM modules (hereon Nokia 12 modules).

The document is divided into the following chapters:

Chapter 2: Introduction

Chapter 3: Nokia 12 module architecture

Introduces the Nokia 12 module architecture, services available in the Nokia 12 module, and terminology used in this document.

Chapter 4: IMlet development environment

Guides you how to program, build, install, run, and debug IMlets using the Nokia 12 module. The required and useful software tools for application development are also presented.

Chapter 5: Java Application Programming Interfaces

Introduces the Java Application Programming Interfaces (APIs) supported by the Nokia 12 module.

Chapter 6: CORBA programming with the Nokia 12 J2ME™ ORB

Introduces the Common Object Request Broker Architecture (CORBA) services available from the Java 2 Micro Edition (J2ME™) Object Request Broker (ORB). These services can be used from an IMlet or remotely over the wireless network.

Chapter 7: Nokia 12 Module ORB services

Describes how to use the CORBA services provided by the Module ORB from an IMlet.

Chapter 8: Example application

Introduces an example application that compiles knowledge and techniques from the other chapters in this document.

Appendix A: Bearer parameters

Describes how to set all the necessary wireless bearer parameters from an IMlet to the Nokia 12 module.

Appendix B: Building tools

Describes how to use the Eclipse Integrated Development Environment (IDE) and Ant building tools to build IMlets.

Appendix C: An example of configuring the Nokia 12 GSM module from IMlet

Describes how to configure Nokia 12 parameters from IMlet.

2. INTRODUCTION

IMlet is a J2ME application that runs on the Information Module Profile (IMP) environment. IMP is a strict subset of the Mobile Information Device Profile (MIDP) commonly used in mobile phones with the distinction that it does not have a User Interface (UI).

IMlets are packed into IMlet Suites. An IMlet Suite is a package – typically a Java Archive (JAR) file – that contains one or more IMlet applications and a manifest file that contains information about the IMlet Suite.

Note: Being familiar with MIDlet programming is helpful as it is very similar to IMlet programming. Majority of documentation available on MIDlet programming, for example via the Internet, also applies to IMlet programming.

This document focuses mainly on how to handle Nokia 12 module -specific issues and those APIs available in the Nokia 12 module that are not covered by Connected Limited Device Configuration (CLDC), MIDP, or IMP. The document also explains issues that are referred to as implementation-specific issues in IMP-1.0. One example of such issue is the Java socket API implementation.

The document also covers Nokia 12 module communications features important to the developer, describes some useful Module ORB services, and briefly explains communications with the Application Module (AM).

CORBA knowledge is required to understand the information in some chapters of this document. The ORB API is used locally for module configuration changes. The required CORBA knowledge for its use is provided in this document.

Because TCP/IP is used in almost all communications, the reader is expected to be familiar with the socket concept. Although the socket API, provided by the J2ME, is quite different from the socket API in the Java 2 Standard Edition (J2SE™), knowledge about Java sockets may be helpful.

Because there is a great deal of information to be covered, the amount of example code in this document is limited to a few rows to keep the document size reasonable. Full example IMlets with source code and build scripts are available in the Nokia 12 Software Development Kit (SDK).

3. NOKIA 12 MODULE ARCHITECTURE

This chapter introduces certain Nokia 12 module hardware and Java Virtual Machine (VM) issues that affect IMlet design and implementation. More information on the Nokia 12 module is available in the *Nokia 12 GSM Module Product Specification*.

Some services in the Nokia 12 module are available for IMlets via Java APIs or CORBA interfaces. Other services are available for external applications from the wireless network or for AMs via CORBA interfaces. These services are listed in Figure 1.

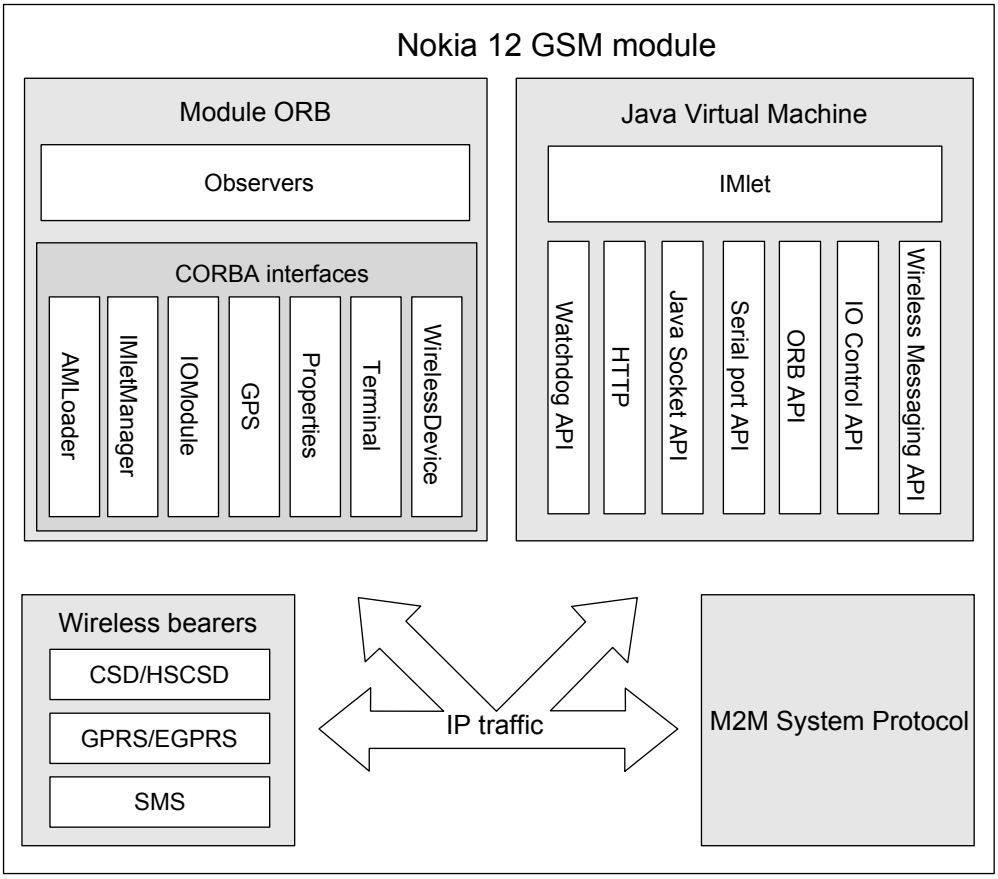

Figure 1. Nokia 12 module architecture

3.1 IP BACKBONE

IP backbone is the basis of communications between different services in the Nokia 12 module. When two services, for example an IMlet and the Module ORB, communicate with each other, they use local loop-back TCP sockets to transmit data. Similarly, IP is used when services are communicating with each other over wireless bearers or with the AM via the M2M System Protocol link.

IP addressing is used to distinguish between network links, and ports are used to identify services within these links. The IP address 127.0.0.1 is always seen as the local address by all services in the Nokia 12 module.

Each service binds to one port to listen to incoming connections. When the client needs to communicate with a service, it simply creates a TCP connection to the target port. After the service has created a server socket, connections to this port can be made from an AM or IMlet, or using a wireless bearer.

The Nokia 12 module shares the same IP address with the AM connected to it. The AM uses the M2M System Protocol 2 socket interface to utilise the IP stack of the Nokia 12 module. This way the AM can communicate with remote entities or the Nokia 12 module. It can also set up server sockets to listen to incoming connections.

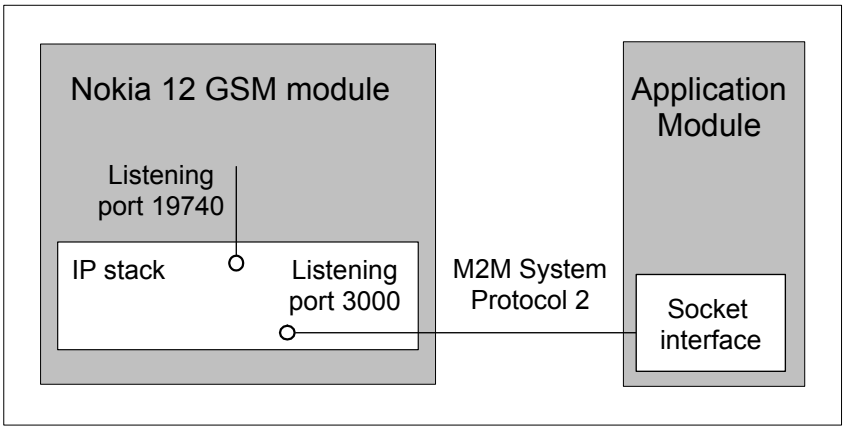

Figure 2. Shared IP stack and example listening ports

Based on the target IP address, the underlying system decides whether to use a wireless bearer, M2M System Protocol 2, or a loop-back socket. If the target address is local, the system checks if a server socket was created from the AM or some other service. If a server socket exists and it was created from the AM, the IP backbone connects to the target port using a socket over the M2M System Protocol link. If the server socket was created from some other service, the IP backbone connects using a local loop-back socket. If the target address is not local, the underlying system automatically uses a wireless bearer to connect to the target.

You can create client and server sockets using the Java Socket API to communicate with applications located in the AM and wireless network. The IMlet can use the J2ME ORB API to communicate locally with the Module ORB services, and remotely with the CORBA services located in a corporate intranet. In both cases, an IMlet can function both as a client and a server. That is, it can initiate communications or respond to service requests initiated from the other end of the application.

For example, an IMlet can listen to incoming connections by creating a server socket and binding this socket to a port known by the AM. The AM can then establish a TCP connection to that port and communicate with the IMlet using TCP sockets. For more information on communicating with the AM, please refer to Chapter 5.

The Nokia 12 module supports all these communications simultaneously. For example, an IMlet and an AM may be communicating with a server application at the same time over a wireless link. Naturally, both share the same network link capacity.

3.2 WIRELESS BEARERS

Wireless bearers are used to communicate over the wireless link with remote servers. Circuit Switched Data (CSD) / High Speed Circuit Switched Data (HSCSD) and General Packet Radio Service (GPRS) / Enhanced General Packet Radio Service (EGPRS) are IP-based bearers and an IMlet can use them via Java sockets or the CORBA API.

An IMlet can use the Short Message Service (SMS) bearer to send and receive short messages using the Wireless Messaging API. It can also use the CORBA services provided by the Module ORB to send and receive short messages.

You can configure wireless bearers with the Nokia 12 Configurator software or using the CORBA API provided by the Module ORB. One of the configured connections must be selected as the default connection. Those services that use wireless bearers generally use the default connection when initiating a connection. Other connections may be configured to handle incoming connections. Some APIs, like the J2ME ORB API, allow an IMlet to choose the bearer to be used also in the IMlet code.

Only one wireless link can be open at a time. Many services in an IMlet or AM can use the same wireless link.

3.3 LOCAL CONNECTIVITY

The Nokia 12 module provides an M2M System Protocol 2 link that allows it to connect with the AM. The M2M System Protocol 2 socket interface is a kind of a remote use API for the TCP/IP and UDP/IP stacks in the Nokia 12 module. The socket API is used from the AM. From an addressing point of view, both the AM and the Nokia 12 module can be seen as one device that share the same IP stack and address. Communications between different internal services are based on IP traffic using local loop-back TCP sockets.

All communication between the Nokia 12 module and AM are done using the socket API. The M2M System Protocol itself is not visible to an IMlet and it is used via other APIs that use the sockets. An IMlet can communicate with the AM using local addresses with the Java socket API or J2ME ORB API.

For more information on the M2M System Protocol 2, please refer to the *Nokia M2M System Protocol 2 Socket Interface User Manual*.

3.4 MODULE ORB

The Module ORB is a CORBA implementation that is built into the Nokia 12 module. It provides services via CORBA interfaces. These services can be used by IMlets, application modules and applications that are located in the wireless network.

Some services provided by the Module ORB have similar functionality to certain Java APIs. These services are provided so that the AM and server applications can use them. For example, the short messaging service at the ET interface provides a very similar functionality as the Wireless Messaging API.

CORBA services offer some extra features like receiving short messages that do not have port numbers. Similarly, Input/Output (I/O) pins can be controlled either by using the IO Control Java API or IO Module CORBA interface. In that case, the CORBA interface is loaded with more features by providing, for example, the possibility to set state listeners for I/O pins.

Please note that the CORBA interface is the only interface available for some services. For example, the Properties interface can be used to change wireless bearer settings and there is no equivalent Java API.

If there is more than one choice available, you can freely choose between the Java API and CORBA interface approaches. The Java API usually provides better performance because CORBA communications for accessing Module ORB services is done using internal sockets.

For more information on the Module ORB services, please refer to Chapter 7.

3.5 JAVA™ VIRTUAL MACHINE

The Java VM in the Nokia 12 module has the following characteristics:

- Available Java heap size of 256 KB
  - At runtime, an IMlet uses memory from the Java heap. The Java VM allocates memory from the Java heap to all dynamically allocated objects.
- Total persistent memory size of 1 MB
  - Total available memory size for all IMlet Suites, JAR files, and Record Management System (RMS) files created by IMlets. Each installed IMlet Suite is stored in this memory.
- Maximum IMlet Suite size of 128 KB

  - Maximum size of one IMlet Suite JAR file. In practice, the maximum number of installed IMlet Suites and IMlets is limited by available persistent memory, but one IMlet Suite cannot be bigger than 128 KB.

- Available RMS storage size of 32 KB per record store
  - Each IMlet Suite can utilize persistent memory using the RMS API. The amount of available memory is limited by the available persistent memory size that is initially 1 MB. One record store is limited to use 32 KB of persistent memory. One IMlet can create multiple record stores and thus use all available persistent memory. Fore more details, please refer to Chapter 5.4.

The number of available Java threads is not artificially limited. The maximum number of threads is limited only by available Java heap memory. Because thread context switching consumes the Central Processing Unit (CPU) resources and memory, avoid creating extensive amounts of threads.

3.5.1 Lifecycle of an IMlet

The Java VM runs one IMlet at a time. Other IMlets and IMlet Suites can be installed but only one IMlet can be active. The Java Application Manager controls the IMlet lifecycle.

Because the Nokia 12 module does not have an UI, a service is required for the installation, running, and removal of IMlet Suites. The Nokia 12 module has a service called IMlet Suite Manager that is used for IMlet Suite management tasks. The IMlet Suite Manager manages IMlet activation, running and shutdown with the Java Application Manager. It also provides CORBA interfaces to remotely managed IMlets.

An IMlet can be activated in two ways: by stopping the currently active IMlet and activating a new one, or by using the ‘force start’ option when activating an IMlet. The Nokia 12 module remembers the last activated IMlet and starts it automatically when the Nokia 12 module is started unless the IMlet has been intentionally stopped. This IMlet is called as the start-up IMlet.

When an IMlet is stopped by the IMlet Suite Manager interface, the Nokia 12 module clears the start-up IMlet record and the IMlet does not start automatically in the next reboot. The IMlet and its IMlet Suite are not removed when the IMlet is stopped.

Before removing an IMlet Suite that contains an active IMlet, the IMlet must be stopped. However, the ‘remove all’ command removes all IMlet Suites regardless of their state, and clears the start-up IMlet record. For more information on IMlet management, refer to Chapter 7.6 and the *Nokia 12 GSM Module Interface Definition Reference Guide*.

	Note: When an IMlet Suite is removed, all records created by that IMlet Suite are also removed.

An IMlet is also stopped if it, or one of the threads created by it, throws an uncaught exception. However, it still remains as the start-up IMlet and starts again after the Nokia 12 module is restarted. The Nokia 12 module does not automatically restart the IMlet if it is stopped by an uncaught exception or a `notifyDestroyed()` method call. You can use the Watchdog API to make sure that the Nokia 12 module and the IMlet will be restarted after the IMlet crashes.

There is also a safety mechanism that clears the start-up IMlet record if the Nokia 12 module reboots 50 times in a row after the IMlet start-up without running the IMlet successfully for more than 30 seconds. The IMlet and its IMlet Suite are not removed from the persistent memory even if the safety mechanism clears the start-up record.

3.5.2 Real Time Clock

The Nokia 12 module supports the Real Time Clock (RTC), but it does not have a built-in battery backup or an API to set the time and date. Instead, the Nokia 12 module uses the GSM Network Indication and Time Zone (NITZ) service. The availability of this service depends on the network.

In IMlet code, the `System.currentTimeMillis()` method returns either the time received from the GSM network or the elapsed time from the Nokia 12 module start-up. When the Nokia 12 module starts, the RTC is started from 1st January 2001. If the NITZ service is available, the RTC date and time are updated according to its values. If time data is not available from the network, it will not be updated. This is also the case when the Nokia 12 module does not have a Subscriber Identity Module (SIM) or is not in network service.

	Note: NITZ is network-specific service. It may take some time before the new time information update arrives after the Nokia 12 module is reset. The time information is network-dependent.

3.5.3 Performance

Java is not designed for real-time applications. Because IMlets run in the Java VM, it is impossible to know exactly how long it takes to execute certain parts of the program. This is mostly due to the Garbage Collection (GC) memory management technique used in the Java VM.

The Java VM in the Nokia 12 module runs at a slightly lower priority level than other services such as communications and the Module ORB. Thus, if the Nokia 12 module is communicating heavily or the Module ORB services are

heavily used by remote applications, an IMlet gets less CPU cycles. This can be avoided by designing a system where all possible services are not used at the same time.

3.6 TEST BOARD

Because the Nokia 12 module does not have a power supply, a SIM card reader, or antenna connectors, a test board is needed for application testing. Nokia provides a test board (1CQ) for development and testing purposes. All examples presented in this document assume that you have access to the Nokia 12 test board or an equally functioning test environment.

Please check that the switches of the Nokia 12 test board are set according to intended use. For example, serial ports can be enabled or disabled by using the Nokia 12 test board switches. For more information on the test board, refer to the *Nokia 12 GSM Module Hardware Integration Manual*.

4. IMLET DEVELOPMENT ENVIRONMENT

This chapter explains how to build IMlets using command line tools from Sun's Wireless Toolkit (WTK).

The overall IMlet creation process goes as follows:

1. Acquire and install necessary tools.
2. Create an IMlet.
3. Compile the IMlet against the **classes.zip** file from the Nokia 12 Concept Simulator.
4. Create a **Manifest.mf** file.
5. Package and pre-verify the IMlet.
6. Test the IMlet in the Nokia 12 Concept simulator.
7. Install the IMlet into the Nokia 12 module using the Nokia 12 Configurator software.

4.1 TOOLS

Required tools

IMlet programming requires the tools listed here. Install the tools in the given order.

- Sun J2SE 1.4 SDK available at http://java.sun.com/j2se/1.4.2/download.html
  - Required to compile Java classes. The **jar.exe** file is required to generate JAR files. Java SDK 1.4.2 or newer is recommended.
- Sun WTK 1.0.4 available at http://java.sun.com/products/j2mewtoolkit/download.html
  - Provides tools for IMlet Suite JAR pre-verification. Make sure that you have version 1.0.4 at you use because that version is intended for the MIDP-1.0 profile. Sun WTK 2 can't be used because it is intended for the MIDP-2.0 profile.
- Nokia 12 IMP 1.0 Concept Simulator available at http://forum.nokia.com/m2m
- Nokia 12 Configurator available at http://forum.nokia.com/m2m
  - Required for Nokia 12 module configuration. The Configurator software also supports IMlet installation and IMlet Suite management.

Useful tools

The tools listed here may be very helpful in IMlet programming.

- Eclipse IDE available at http://www.eclipse.org
  - Eclipse is an open source IDE for Java development. For more information on using Eclipse in IMlet programming, please refer to Appendix B: Building tools.
- Ant build tool available at http://ant.apache.org
  - Needed to process Ant build scripts. For more information on using Ant in IMlet programming, please refer to Appendix B: Building tools.
- Antenna available at http://antenna.sourceforge.net
  - Provides Ant tasks to automatically generate IMlet Suite JAR files from IMlet classes. The Antenna tool can construct **Manifest.mf** files automatically and handle pre-verification with the **preverify.exe** from Sun WTK. It can be used together with an Ant-based IDE such as Eclipse. For more information on using Antenna in IMlet programming, please refer to Appendix B: Building tools.
- Nokia 12 SDK available at http://forum.nokia.com/m2m
  - Nokia 12 SDK has many example IMlets and example build scripts ready to be used with Ant, Eclipse and Antenna.

Any other IDE that supports the creation of MIDlets or IMlets can also be used. The most important step when using an IDE is to take the **classes.zip** file from the Nokia 12 Concept Simulator and configure the IDE to use those classes as system classes. The **classes.zip** file is required to obtain all possible APIs available for IMlet.

Note: You must compile all IMlet Suites against the **classes.zip** file.

4.2 CREATING AN IMLET

One class must inherit the `javax.microedition.midlet.MIDlet` class. This is the IMlet class and it functions as the application starting point. It must have the methods listed below.

```
public void startApp() throws MIDletStateChangeException
```

This is the IMlet start-up point. The Java Application Manager calls this method after an IMlet is loaded into the memory to start its execution. It can be

compared to the `main()` method in the J2SE application. The method can be called multiple times; first when the IMlet is started and then every time an IMlet is resumed from the paused state.

```
public void pauseApp()
```

The Java Application Manager calls this method to notify the IMlet that it will be set to the paused state.

```
public void destroyApp(boolean unconditional) throws
MIDletStateChangeException
```

The Java Application Manager calls this method when the IMlet is to be destroyed. If the unconditional flag is set to 'false', the IMlet can prevent the destroy request by throwing a MIDletStateChangeException. If the IMlet calls the `notifyDestroyed()` method, the Java Application Manager does not call the `destroyApp()` method.

When the Java Application Manager starts the IMlet, it first calls the IMlet's constructor and then the `startApp()` method. Your code should start from the `startApp()` method.

The IMlet can stop execution by calling the `notifyDestroyed()` method from the `MIDlet` class.

4.3 COMPILING AN IMLET

To ensure that all APIs used by the IMlet match with those located in the Nokia 12 module, you must compile the classes against the **classes.zip** file provided with the Nokia 12 Concept Simulator.

IMlets compiled against other system class libraries, such as **classes.zip** from the Sun WTK reference implementation, may not run in the Nokia 12 module. At least not all APIs are available if the wrong **classes.zip** file is used.

4.4 CREATING A MANIFEST FILE

A manifest file contains important information on the IMlet name and start-up class name. The Java Application Manager needs this information when starting the IMlet.

An example of a Manifest.mf file

```
Manifest-Version: 1.0
MicroEdition-Configuration: CLDC-1.0
MIDlet-Name: ExampleIMlets
MIDlet-Vendor: Nokia
MIDlet-1: Blinker, , com.nokia.m2m.sdk.examples.iocontrol.Blinker
MIDlet-Version: 1.0.0
MicroEdition-Profile: IMP-1.0
```

```
My-Server-Port: 8080
```

Manifest files created with different tools can vary slightly, but the example manifest file given here lists all the mandatory elements and one application property. The most important row is the row starting with 'MIDlet-1'. This row tells the Java Application Manager which class to load when an IMlet called Blinker is started. The above manifest file is used in the example build process presented in Chapter 4.8.

An application property is used to store application properties so that there is no need to hardcode them to the IMlet code. Application properties can be read from the IMlet code by using the `getAppProperty()` method from the MIDlet object. Because a Java Application Descriptor (JAD) file is not installed into Nokia 12 module, the only place to store application properties is the manifest file.

4.5 PACKAGING AND PRE-VERIFYING AN IMLET

After creating a manifest file, you must make a JAR file that contains the IMlet class, manifest file, and all other classes needed by the IMlet. All classes packaged into the IMlet Suite JAR file must be pre-verified using the **preverify.exe** tool provided by the WTK 1.0.4.

A JAD file that can be created by many development tools is not needed when creating IMlets for the Nokia 12 module. All needed information is stored in the manifest file located inside the JAR file.

Many IDE tools used for MIDlet programming automatically create MIDlet Suites using the MIDP 1.0 profile in the manifest file. The IDE tool must thus be configured to use the IMP 1.0 profile instead of the MIDP 1.0 profile.

4.6 DEPLOYING AND RUNNING AN IMLET

After the IMlet Suite JAR file is created, the IMlet can be installed into the Nokia 12 module using the Nokia 12 Configurator. The Nokia 12 Configurator enables the installation of the IMlet Suite into the Nokia 12 module, selection of an IMlet from those available in the IMlet Suite, and starting and stopping of the selected IMlet.

The Nokia 12 Configurator also provides a small-scale Over-the-Air (OTA) IMlet installation functionality using the CSD bearer. This allows you to update IMlets remotely into already installed Nokia 12 modules.

The Nokia 12 module also offers an IMlet Suite Manager service, which can be used to manage IMlets remotely. See the *Nokia 12 GSM Module Software Developer's Guide* for more information.

4.7 DEBUGGING AN IMLET

You can debug IMlets using the Nokia 12 Concept Simulator. The simulator displays the state of output pins and provides an UI to change the input pin states.

You can use the serial port available in the Nokia 12 module to print debug information and use a PC to display it. The `com.nokia.m2m.sdk.examples.logger.SerialPortLogger` class is provided in the Nokia 12 SDK as an example of how to use the serial port to display log information. If the serial port is needed for some other use, the debug information can be stored into RMS files. Also output pins can be used to display IMlet status information.

When the `com.nokia.m2m.sdk.examples.logger.SerialPortLogger` class is used with a PC application, for example the Windows HyperTerminal, set the serial port options as described in Table 1.

Table 1. Serial port options

Option	Value
Port speed	115 200
Data bits	8
Parity	None
Stop bits	1
Flow control	None

Note: The Nokia 12 Concept Simulator does not support some of the features available in the Nokia 12 module. The only way to debug these features is to run the IMlet in a Nokia 12 module. See the Nokia 12 Concept Simulator release note for list of the features that the Nokia 12 Concept Simulator does not support.

4.8 AN EXAMPLE OF BUILDING AN IMLET WITHOUT INTEGRATED DEVELOPMENT TOOLS

The example given in this chapter describes how to build an IMlet without using any integrated development tools. The example is for Windows, and it supposes that all required tools are found from the system path.

The example IMlet uses the IOControl API to set the output pin 7 on and off. If the Nokia 12 module is attached to the Nokia 12 test board, the led attached to output pin 7 starts to blink.

```
package com.nokia.m2m.sdk.examples.iocontrol;
```

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;

import com.nokia.m2m.imp.iocontrol.IOControl;

/**
 * Sets the output pin 7 on and off.
 */
public class Blinker extends MIDlet {

   private static final int OUTPUT_PORT = 7;

   protected void startApp() throws MIDletStateChangeException {

      boolean on = true;
      try {

         if (on) {
            on = false;
            IOControl.getInstance().setDigitalOutputPin(OUTPUT_PORT,
true);
         } else {
            on = true;
            IOControl.getInstance().setDigitalOutputPin(OUTPUT_PORT,
false);
         }

         Thread.sleep(500);

      } catch (Exception ignored) {

      }

   }

   protected void pauseApp() {
      // Not needed.
   }

   protected void destroyApp(boolean unconditional) throws
MIDletStateChangeException {
      // Not needed.
   }
}
```

To build an IMlet:

1. Compile an IMlet using the **javac.exe** file from the Java SDK. Use the **classes.zip** file from the Nokia 12 Concept Simulator as the system library. The prerequisite for the following command is that the **classes.zip** file is found from the **lib** directory. The generated classes are stored into the **classes** directory.

```
javac -bootclasspath lib/classes.zip -d classes -sourcepath src
src/com/nokia/m2m/sdk/examples/iocontrol/Blinker.java
```

2. Pre-verify the generated classes. Use the **classes.zip** file from the Nokia 12 Concept Simulator in the classpath that is given to the **preverify.exe** file. The following command pre-verifies all classes located in the **classes** directory and generates pre-verified class files into the **preverified** directory:

```
preverify -classpath lib/classes.zip -d preverified classes
```

3. Package all the pre-verified classes into the **blinker.jar** file, and add the **Manifest.mf** file into the JAR file. The example manifest file described in Chapter 4.4 is used as the **Manifest.mf** file here.

```
jar cmf Manifest.mf blinker.jar -C preverified com
```

4. The **blinker.jar** file is now ready to be installed and run in the Nokia 12 module. Use the Nokia 12 Configurator to install the IMlet into the Nokia 12 module and start the IMlet after the installation is completed.

5. Run the Blinker IMlet in the Nokia 12 Concept Simulator.

6. The Nokia 12 IMP 1.0 Concept Simulator requires that the IMlet Suite includes an associated JAD file. Before the IMlet can be tested in the simulator, the following **blinker.jad** file is required:

```
MIDlet-1: Blinker, , com.nokia.m2m.sdk.examples.iocontrol.Blinker
MIDlet-Jar-Size: 2159
MIDlet-Jar-URL: blinker.jar
MIDlet-Name: Blinker
MIDlet-Vendor: Nokia Corporation
MIDlet-Version: 1.0
```

7. The MIDlet-Jar-Size value indicates the size of the **blinker.jar** file. It is not used by the Nokia 12 IMP 1.0 Concept Simulator.

8. The following command sequence runs the IMlet in the Nokia 12 IMP 1.0 Concept Simulator:

```
SET
EMULATOR_HOME=C:\USERS\WTK104\wtklib\devices\Nokia_12_IMP_1_0_Concept_Simul
ator
SET DESCRIPTOR FILE=blinker.jad

java -classpath
%EMULATOR_HOME%\bin\emulator.jar;%EMULATOR_HOME%\externalapp\classes;%EMULA
TOR_HOME%\bin -Demulator.home=%EMULATOR_HOME% com.nokia.phone.sdk.Emulator
-Xdescriptor %DESCRIPTOR_FILE%
```

9. Change the EMULATOR_HOME environment variable according to your system.

5. JAVA APPLICATION PROGRAMMING INTERFACES

This chapter describes the APIs that are available through the Java interface.

5.1 CONNECTOR FRAMEWORK

This chapter explains the connector framework and protocol schemes supported by the Nokia 12 module. The connector framework is covered only briefly; more information is available from MIDP 1.0 documentation.

The connector framework provides one entry point to create multiple different connections. Majority of communication protocols supported by the Nokia 12 module are used through the connector framework.

Only some classes in the connector framework and some of their methods are introduced in this section. All the classes are located in the `javax.microedition.io` package. A more detailed API specification can be found in MIDP 1.0 and IMP 1.0 Javadocs.

5.1.1 Connector class

The `javax.microedition.io.Connector` class creates connections. It provides a few methods to create `Connection` objects from an URL string and some optional arguments.

```
static Connection Connector.open(String url)
```

This method creates a `Connection` object using an URL string and default mode settings. The connection is created using the READ_WRITE mode, and both data sending and receiving are possible. More information on protocol-specific limitations to this rule can be found in Chapters 5.1 - 5.6.

```
static Connection Connector.open(String url, int mode)
```

This method creates a `Connection` object using an URL string and specified mode settings. Available modes include:

- Connector.READ
- Connector.WRITE
- Connector.READ_WRITE

If the connection is opened in the READ mode, only data receiving is possible from that connection.

`static Connection Connector.open(String url, int mode, boolean timeout)`

This method creates a `Connection` object using a specified timeout. If a timeout is given and the timeout value is reached, method calls from the `Connection` object can throw an InterruptedIOException. All protocol implementations may not support the timeout feature. More information on timeout support in different protocols can be found in the protocol specific Chapters 5.1 - 5.6.

The `open()` method in the `Connector` returns a protocol-specific `Connection` object that is then used to transmit data. Before data transmission, the returned `Connection` object must be typecast to a protocol-specific implementation object. Class of the returned `Connection` object is specified by the protocol scheme in the URL given in the `open()` method.

Table 2 shows all the protocol schemes supported by the Nokia 12 module, and the returned `Connection` class for each supported protocol.

Table 2. Supported protocol schemes

Scheme	Protocol	Connection class	Description and example URL
http	HTTP	HttpConnection	HTTP connection over a TCP bearer. http://www.example.com:80/index.html
socket	TCP	StreamConnection	TCP client connection. socket://10.35.1.195:9000 socket://example.com:80
socket	TCP	StreamConnectionNotifier	TCP server connection for receiving incoming TCP connections socket://:9000
datagram	UDP	DatagramConnection	UDP socket for sending and receiving UDP datagrams. datagram://10.35.1.195:9000 datagram://:9000
comm	Serial port	StreamConnection	Stream connection for sending and receiving data using a built-in serial port. comm:3;baudrate=115200
sms	SMS	MessageConnection	Message connection for sending and receiving short messages. sms://+0123xxxx sms://:3000

5.1.2 Connection lifecycle

When a connection is created, the Nokia 12 module associates some native resources, for example a serial port or TCP sockets, with that connection. Some of the resources are numbered. When a connection using a resource is open, other attempts to create the same type of connection result to a `java.io.IOException`.

Generally, you should use the `java.io.InputStream` and `java.io.OutputStream` classes so that only one thread uses the same stream at a time. If multiple threads are using the same stream, you should implement a synchronisation structure to ensure data integrity.

All streams and connections must be closed when they are not needed any more. If connections are simply left to Java GC, the reserved resources may be unavailable for the IMlet until the Nokia 12 module is restarted.

5.2 IP CONNECTIONS

The `StreamConnection` and `DatagramConnection` classes enable communication using UDP datagrams and TCP sockets. The Nokia 12 module supports communication between an IMlet and remote targets over wireless bearers. Local TCP sockets over the M2M System Protocol 2 link can also be used for communication between an IMlet and an application module.

Both GPRS/EGPRS and CSD/HSCSD bearers are supported. When a wireless bearer is used, the Nokia 12 module uses a Point-to-Point Protocol (PPP) link to transmit data over the wireless network. Before TCP or UDP connections can be used, one CSD or GPRS bearer must be configured into the Nokia 12 module and set as the default connection. Configuration can be done using the Nokia 12 Configurator.

TCP and UDP sockets are used via the Connector interface. These sockets are divided into client sockets and server sockets. Client sockets are used to initiate communications from an IMlet and server sockets are used to receive connections initiated by remote hosts. Client and server sockets can be used simultaneously. One IMlet can have multiple sockets open simultaneously and multiple threads running and serving these sockets.

Table 3. URL syntax for IP connections

URL	Description	Link
socket://10.35.1.5:900 socket://example.com:80 socket://127.0.0.1:9000	TCP client socket that is connected to a given target address and target port.	A wireless link is created if the target address is not 127.0.0.1. The link stays open until the last socket is closed.
socket://:900	TCP server socket that listens to port 900.	This call does not open a wireless link.

URL	Description	Link
datagram://10.35.1.5:9000 datagram://example.com:9000	UDP client socket used for sending and receiving datagrams.	A wireless link is created if the target address is not 127.0.0.1. The link stays open until the socket is closed.
Datagram://:900	UDP server socket that listens to port 900 and is used for receiving and sending datagrams.	This call does not open a wireless link.

The socket API supports both IPv4 IP address and host name addressing, but IPv6 addressing is not supported. If hostnames are used, the Internet name server settings must be negotiated for the PPP link between the Nokia 12 module and the Internet service provider. If an IP address is used, it must be given in a dotted decimal format. A local host address, 127.0.0.1, is used to create local connections. Local connections can be used to communicate with the AM.

The number of open sockets is limited to ten. This is the total limit for all socket types: TCP, UDP, client, and server. If an IMlet tries to open more that ten sockets, the `open()` method of the `Connector` class results in a java.io.IOException. If a server socket receives a new connection request when in a listening mode, and the maximum number of open sockets has already been reached, the Java VM immediately closes the accepted connection. This limitation only applies to the Java socket API that is used via the connector framework. Other services in the Nokia 12 module that function over TCP/IP (like the J2ME ORB) are not affected by this limitation.

5.2.1 Wireless link

The Nokia 12 module manages the wireless link automatically. When a client connection is created and connected to a non-local target, the Nokia 12 module creates a wireless link for the connection and then creates a connection over this link. The default connection is used for the link creation. This connection can be set and changed using the Nokia 12 Configurator. A CSD or GPRS bearer must be set as the default connection bearer.

If the first attempt of creating a link fails, the socket API automatically makes another connection attempt before raising a java.io.IOException. This means that in the worst case, one `open()` method call can result to two data calls or two GPRS context activation attempts.

When the last socket is closed, the link is also closed. However, other services, like the Module ORB or AM, can keep the link open even if the IMlet that used the socket API closes its last socket. The basic rule for link management is that the entity that has created the link and initiated communications is responsible

for closing the link. Chapters 5.2.2 and 5.2.3 explain the link management with different types of sockets in more detail.

The link is closed in the following situations:

- The last socket is closed and no other service in the Nokia 12 module needs the link anymore.
- The link inactivity timeout has been reached. This Connection timeout parameter can be set for the wireless bearer using the Nokia 12 Configurator. The timer monitors the wireless link and detects if no data has been sent or received within a given time period.
- The remote peer drops the link.
- A network error causes the link to drop.

The IMlet must always close all sockets it has created or accepted. Sockets must not be left to the Java GC, because this may result the IMlet to quickly run out of available sockets. All opened streams must also be closed.

The IMlet must first close the opened streams and then the socket. The IMlet must close the open connection even when the remote peer first closes the connection.

5.2.2 StreamConnection

The `StreamConnection` represents a TCP connection – local or remote – between an IMlet and another application. When the IMlet opens a `StreamConnection` to the server, the created connection is called the client connection. When the IMlet creates a `StreamConnectionNotifier` and listens for an incoming connection request, the connection is called the server connection (also known as server socket).

5.2.2.1 StreamConnection client

Creating a StreamConnection

The `Connector` protocol 'socket' is used to create a TCP connection to a remote host. The code example shown below opens a connection to address 10.35.1.5:900. This initiates the build-up of a wireless network link.

```
StreamConnection c =
        (StreamConnection) Connector.open("socket://10.35.1.5:900");
```

First, a wireless link is created (if one is not already open). Then, a TCP connection is created to the given target address over the link. After the

connection is created, the link stays open until the socket is closed or the remote end closes the link. If the IMlet creates other sockets, they will automatically share the same link. The link stays open until the last socket is closed.

Connector parameters

The Nokia 12 module does not use the timeout argument. All `read()` methods in the `StreamConnection` block until the connection is closed or data is returned. The only way to interrupt a blocked read is to close the socket. The code example below is equivalent to the previous code example.

```
StreamConnection c =
      (StreamConnection) Connector.open( "socket://10.35.1.5:900",
Connector.READ_WRITE, true);
```

If the IMlet needs to implement a non-blocking reading, it can use the `available()` method from the `InputStream` to check if the stream has data available before calling the `read()` method.

Using a hostname

The following code example uses a hostname as the target address instead of an IP address. When a hostname is used in the address string, the Nokia 12 module first opens the wireless link. It then resolves the target IP address by performing a Domain Name Server (DNS) lookup. Finally it opens a connection to the target. The Internet name server settings must be negotiated for the wireless link if hostnames are used as the target address.

```
StreamConnection c =
         (StreamConnection)
Connector.open("socket://host.example.com:900");
```

TCP client example IMlet

The example IMlet is located in the `com.nokia.m2m.sdk.examples.socket` package provided in the Nokia 12 SDK.

The following code example sends data to a server using the `OutputStream` and reads the response from the `InputStream`. After the response is received, the program closes all the opened streams and created `StreamConnection` instances. The important part in this example is the `finally{}` block that ensures proper clean-up in all situations.

```
public class TCPClient extends MIDlet {
```

```
protected void startApp() throws MIDletStateChangeException {
   log("TCPClient starting");

   StreamConnection conn = null;
   OutputStream out = null;
   InputStream in = null;

   try {

      String msg = "HELLO\n";

      // Wait a while for the Nokia 12 module to get into network
      // service after boot-up.
      Thread.sleep(20000);

      log("Connecting...");
      conn =
         (StreamConnection) Connector.open("socket://10.35.1.195:9001");
      log("Connected");

      // Open streams for sending and receiving.
      out = conn.openOutputStream();
      in = conn.openInputStream();

      log("Sendign request");
      // Send a message to the server.
      out.write(msg.getBytes());

      // Read the response.
      log("Reading response");
      byte[] responseBytes = new byte[msg.length()];

      int offset = 0;
      int ret = 0;
      int length = msg.length();

      //
      // The read() call may return less data than was
         enquired.
      //

      while (offset < length) {
         ret = in.read(responseBytes, offset, length - offset);
         if (ret == -1) {
            // Connection is closed by the server.
            log("Error: connection closed");
            return;
         } else {
            offset += ret;
         }
      }

      // Check the response.
      String response = new String(responseBytes);

      if (response.equals(msg)) {
         log("OK! server returned:" + response);
      } else {
         log("FAIL! invalid response:" + response);
      }
   } catch (Exception e) {
      log("Exception: " + e.toString());
   } finally {

      // Write clean up code in the finally block to
      // make sure that all resources are released.

      // Close all open streams and connections.
```

```
      if (in != null) {
            try {
               in.close();
            } catch (Exception ignored) {
            }
         }
         if (out != null) {
            try {
               out.close();
            } catch (Exception ignored) {
            }
         }

         // Close the connection.
         if (conn != null) {
            try {
               conn.close();
            } catch (Exception ignored) {
            }
         }

      }

   }
```

The `Thread.sleep()` method is used to wait for a network registration after start-up. The time between starting the Nokia 12 module and having the network registration ready may vary in different GSM networks. The IMlet can use the `SignalQuality` parameter in CORBA API to monitor the network registration status before it tries to communicate over the wireless bearer. Another way to wait for network registration is to try to open a connection, catch the raised exception, commit little sleep, and try again until the connection succeeds.

5.2.2.2 StreamConnectionNotifier

The `StreamConnectionNotifier` is used to accept incoming TCP connection requests. The `acceptAndOpen()` method in the `StreamConnectionNotifier` blocks until a connection is accepted. When the `StreamConnectionNotifier` accepts a new connection, it creates and returns a `StreamConnection` to the IMlet. The returned `StreamConnection` can be used to send and receive data in the same way as with the client connection described in Chapter 5.2.2.1. The IMlet is responsible for closing the accepted connection when it is not needed anymore.

TCP server example IMlet

The example IMlet is located in the `com.nokia.m2m.sdk.examples.socket` package. It functions as a server that listens to incoming connections. When the client connects to the IMlet and sends data, the IMlet echoes the received data back to the sender.

The following code example creates a `StreamConnectionNotifier` to listen to port 900. After initialisation, it calls the `acceptAndConnect()` method to accept

incoming connection requests. This method blocks until a connection request arrives. However, no wireless link is created when this code is executed. The client is responsible for opening the network link by creating a CSD call or by opening a GPRS context via the Wake-Up Service (WUS) functionality. See the *Nokia 12 Software Developer's Guide* for more information about the WUS.

You can test the IMlet with a TCP client program, for example, a Telnet application.

```
public void startApp() throws MIDletStateChangeException {

   StreamConnectionNotifier notifier = null;
   InputStream in = null;
   OutputStream out = null;
   StreamConnection conn = null;

   // Use a buffer to copy data.
   byte[] buffer = new byte[1024];

   try {

      notifier =
         (StreamConnectionNotifier) Connector.open("socket://:900");

      while (true) {

         // Sets the socket into the listening mode.
         // The call blocks until
         // a new connection is accepted.

         conn = notifier.acceptAndOpen();

         try {

            in = conn.openInputStream();
            out = conn.openOutputStream();

            //
            // Read data in 1024 byte chunks. Avoid using the
            // int InputStream.read() reading because it
            // causes a native read for each byte.
            //

            //
            // The read() call may return
            // less data than was enquired.
            // read() returns the number of bytes received
            //
            int ret = 0;
            while (ret != -1) {

               //
               // Read the data. Returns -1 when the connection is
               // closed by the remote peer.
               // The method blocks until
               // something is received.
               //
               ret = in.read(buffer, 0, buffer.length);

               if (ret == -1) {
                  // The connection is closed by the server.
                  log("connection closed");
               } else {
                  out.write(buffer, 0, ret);
               }
```

```
            }

         } catch (Exception e) {

            // This exception can be
            // caused by an unexpected link drop.

            log("Exception while transmitting " + e.toString());
         } finally {

            //
            // Make sure that the streams are closed before
            // going to the next round.
            //
            if (out != null) {
               try {
                  out.close();
               } catch (Exception ignored) {
               }
            }

            if (in != null) {
               try {
                  in.close();
               } catch (Exception ignored) {
               }
            }

            if (conn != null) {
               try {
                  conn.close();
               } catch (Exception ignored) {
               }
            }

            in = null;
            out = null;
            conn = null;

         } // try

      } // while

   } catch (Exception e) {
      log(e.toString());
   }

}
```

Because creating a `StreamConnectionNotifier` does not open a link or create a connection to the remote target, the `open()` method in the `Connector` throws an exception only when another socket has reserved the selected port.

When a socket has been accepted this way, the Nokia 12 module keeps the link open until the remote peer closes the link or the last socket has been closed. A separate thread is usually created to serve the accepted connection and the IMlet's main thread is left to listen for more connections. The clean-up done in the `finally{}` block is very important in this example because the IMlet is designed to stay in an accept-loop forever. Without this kind of error handling, the IMlet will eventually run out of sockets.

The Nokia 12 module does not use the timeout argument in the `open()` method. The `acceptAndOpen()` method blocks until a connection is accepted or the `StreamConnectionNotifier` is closed.

5.2.3 DatagramConnection

The `DatagramConnection` can be used to transmit UDP datagrams between an IMlet and a remote peer.

The maximum size for sent and received datagrams is 1240 bytes. If the IMlet tries to send longer datagrams, the `send()` method throws a java.io.IOException. If the remote peer sends a longer datagram to the Nokia 12 module, it is discarded.

Please note that UDP is a connectionless and unreliable protocol. In situations with a great deal of network congestion, a big percentage of transmitted datagrams may be lost during transmission. Neither UDP nor the Nokia 12 module has any means to inform the protocol user about lost packets. An IMlet can send and receive datagrams only by using wireless bearers. Sending datagrams to local targets, such as the AM, or receiving datagrams from local targets is not supported.

The `send()` method in the `DatagramConnection` throws a java.io.IOException only if the `DatagramConnection` is closed or the network link drops. A successful return from the `send()` method does not necessarily mean that the datagram has reached the designated target. It just means that it was successfully sent.

5.2.3.1 DatagramConnection in the client mode

When the `DatagramConnection` is used in the client mode, it can be used to communicate with one target. The socket that is created this way cannot be used to send datagrams to or receive them from other addresses. This kind of datagram socket is known as a connected datagram socket.

DatagramClient example IMlet

```
public void startApp() throws MIDletStateChangeException {

      DatagramConnection conn = null;

      try {

         //
         // Open a connection to the target. This call
         // opens the wireless link.
         //
         conn =
            (DatagramConnection) Connector.open(
               "datagram://10.35.1.195:9000");

         //
```

```
            // Send "HELLO" to the server.
            //
            Datagram d;
            byte[] data = "HELLO".getBytes();
            d = conn.newDatagram(data, data.length);

            conn.send(d);

            // Allocate room to receive the response.
            Datagram resp = conn.newDatagram(1024);

            //
            // Receive the response.
            // The method blocks until the datagram is received or the link
            // is dropped.
            //
            conn.receive(resp);

            data = resp.getData();

         } catch (Exception e) {
            log(e.toString());

         } finally {
            if (conn != null) {
               try {
                  conn.close();
               } catch (Exception ignored) {
               }
            }
         }

      }
```

If the wireless link is not open when the `open()` method in the `Connector` is called, it is created before the method returns. Because UDP is a connectionless protocol, no UDP data is transmitted in the `open()` phase. The first UDP data transmission is done on the `send()` row.

The `receive()` method blocks until one datagram is received or the network link drops. If the network link drops when the thread is blocked in the `receive()` method, the method throws a java.io.IOException. This exception is also raised if the `receive()` method is called for the client mode `DatagramConnection` that has lost the network link.

No timeout has been implemented for the `receive()` method call. Another thread can close the DatagramConnection and free the thread that was blocked in the `receive()` call. This way the `java.util.Timer` and `java.util.TimerTask` classes can be used to implement a timeout feature.

5.2.3.2 DatagramConnection in the server mode

The primary use of the server mode `DatagramConnection` is to listen for incoming datagrams, process them, and return a response to the sender.

The network link control works similarly as with the `StreamConnectionNotifier`. The network link is not automatically created

when the server mode `DatagramConnection` is created. If the remote end, for example a CSD call, initiates the link, it will be kept open by the Nokia 12 module until all sockets are closed or the remote end closes the link.

The server mode `DatagramConnection` can be used for datagram sending if the network link is open. The `receive()` method blocks until the datagram is received and the `send()` method throws a java.io.IOException if the network link was closed.

The `Connector` URL for the server mode `DatagramConnection` is datagram://:port. Because creating a server mode `DatagramConnection` does not open the network link, the `open()` method throws a java.io.Exception only if the URL syntax is invalid or the given port is already reserved for another `DatagramConnection`.

DatagramServer

The example IMlet is located in the `com.nokia.m2m.sdk.examples.socket` package.

The following code example creates a server mode `DatagramConnection` to port 900. After the socket is created, the IMlet starts receiving and processing datagrams. The example code echoes the received datagrams back to the sender.

```
public void startApp() throws MIDletStateChangeException {

      DatagramConnection dc = null;

      try {

         log("DatagramServer starting");

         //
         // Create a server socket to listen to the port.
         //

         dc = (DatagramConnection) Connector.open("datagram://:900");

         while (true) {

            Datagram dgram = dc.newDatagram(1200);

            // The call blocks until the datagram is reveived.
            dc.receive(dgram);

            // ... Process received data here ...

            //
            // Set the sender address as the new target address and
            // send the datagram back to the sender.
            //
            dgram.setAddress(dgram.getAddress());

            try {
               // Throws an IOException if the link is not open.
               dc.send(dgram);
            } catch (IOException ignored) {
```

```
            }

         }

      } catch (Exception e) {
         log(e.toString());
      } finally {
         if (dc != null) {
            try {
               dc.close();
            } catch (Exception ignored) {
            }
         }
      }

   }
```

Sending back data is possible because the link should still be open when the response is sent. If the link is closed when the `send()` method is called, the method throws a `java.io.IOException` that is caught in the code and ignored.

5.3 HTTP API

The HTTP API can be used to create HTTP requests to and receive responses from a remote server. The Nokia 12 module has separate settings for the HTTP API. HTTP can only be used with CSD and GRPS bearers. It is not possible to use it to communicate with the AM.

Because the HTTP API uses its own network connection settings and wireless network link management, it is not possible to use it together with the CORBA or Java Socket APIs. Before using the HTTP API, make sure that you have closed all TCP and UDP sockets and that the J2ME ORB is shut down by the `orb.destroy()` method call.

The HTTP API opens a wireless link when the `connection.getInputStream()` method is called for the first time for the connection. The wireless link is closed when the HTTP connection is closed. Multiple HTTP requests can be done over the same wireless network link.

Wireless bearer settings for HTTP connections are set using the Nokia 12 Configurator or the Wireless Device CORBA API. For more information on changing the configuration using the CORBA API, please refer to Chapter 6.

An example of a HTTP request

The following example creates a HTTP request to the server and prints the response to a serial port. This example IMlet is available in the Nokia 12 SDK.

```
package com.nokia.m2m.sdk.examples.http;

import java.io.DataInputStream;
import java.io.OutputStream;
```

```
import javax.microedition.io.Connector;
import javax.microedition.io.HttpConnection;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;

import com.nokia.m2m.sdk.examples.logger.SerialPortLogger;

/**
 * IMlet demonstrates the HTTP connection usage.
 *
 */
public class HTTPIMlet extends MIDlet {

   public void startApp() throws MIDletStateChangeException {

      // Address and port for the server.
      final String HOST = "10.35.1.195";
      final int PORT = 8080;

      HttpConnection conn = null;
      DataInputStream in = null;
      OutputStream out = null;

      log("HTTPImlet starting");

      //
      // When a HTTP connection is asked from the
      // Connector, the used syntax can be
      //
      // "http://ip:port/resource"
      //  or with host name
      // "http://www.example.com/".
      //

      // Get the root page /index.jsp
      String url = "http://" + HOST + ":" + PORT + "/index.jsp";

      try {

         // Wait for a while to get the network service.
         log("Waiting for 30 seconds");
         Thread.sleep(30000);

         log("Creating connection " + url);
         conn = (HttpConnection) Connector.open(url, Connector.READ);
         conn.setRequestMethod(HttpConnection.GET);

         // A connection is created and HTTP request sent.
         log("Sending request");
         in = conn.openDataInputStream();

         // The HTTP server sends a response with the
         // status code 200 if the HTTP request succeeded.
         int status = conn.getResponseCode();
         log("Status= " + status);

         long len = conn.getLength();
         log("Content lenght= " + len);

         StringBuffer buff = new StringBuffer();
         char ch = 0;

         while (ch != -1) {
            ch = (char) in.read();
            buff.append(ch);
         }

         // Print the received response.
```

```
            log("Server returned:" + buff.toString());

         } catch (Exception e) {


            log(e.toString());

         } finally {

            // This block is always executed.
            // Release all resources.

            if (conn != null) {
               try {
                  conn.close();
               } catch (Exception ignored) {
               }
            }

            if (in != null) {
               try {
                  in.close();
               } catch (Exception ignored) {
               }
            }

            if (out != null) {
               try {
                  out.close();
               } catch (Exception ignored) {
               }
            }
         }
      }

      private void log(String msg) {
         SerialPortLogger.getInstance().write(msg);
      }

      public void pauseApp() {
      }

      public void destroyApp(boolean arg0) throws MIDletStateChangeException {
      }

}
```

5.4 RECORD MANAGEMENT SYSTEM

This chapter explains briefly the RMS in the Nokia 12 module. For more information on RMS, please refer to MIDP 1.0 documentation.

IMP is a subset of the MIDP, and MIDP defines a set of classes for storing and retrieving data. These classes are called the RMS. With RMS, it is possible to store data across invocations of an IMlet. Different IMlets in the same IMlet Suite can also use the RMS to share data, but data is not available for IMlets located in the other IMlet Suites.

Tip: It is recommended that you familiarise yourself with using the RMS API by reading the RMS documentation available from Sun's web pages.

The memory space available for each record store in the Nokia 12 module is 32 KB. One IMlet Suite can use multiple record stores. Because all persistent memory space is shared between JAR files and their record stores, no room will be left for the record stores if the Nokia 12 module is full of IMlets.

To use the RMS, import the `javax.microedition.rms` package.

Recommended use of the RMS

Changes to the record store are kept in the RAM until the record store is closed. After that, the information is written into permanent memory. This is why the record store should be open only when necessary and closed as soon as possible after use. This also reduces the possibility of file corruption.

```
// Open the record store.
rs = RecordStore.openRecordStore(rsName, true);
// Use the record store
rs.addRecord(b, 0, b.length);
// Close the record store.
rs.closeRecordStore(rsName);
```

Recommended procedure in case of a power loss

In case of a power loss, the RMS file may be corrupted and data in the record damaged. Also, you may not be able to open the data record again. In that case, the following procedure is recommended:

```
// Open RMS, throws RecordNotFoundException
try {
   rs = RecordStore.openRecordStore(rsName, true);
   // If the record store is corrupted, continue as follows
} catch (Exception e) {

   // Invalid record store contents.
   try {
      //Delete the corrupted record store.
      RecordStore.deleteRecordStore(rsName);
   } catch (Exception ignored) {
   }

   try {
      // Re-open the file.
      rs = RecordStore.openRecordStore(rsName, true);
   } catch (Exception ignored) {
   }
}
```

The above code prepares to catch an exception when it tries to open the record store and removes the record store if it is corrupted. The RMS API does not allow the IMlet to create a new file using the same name before the corrupted file is removed.

5.5 SERIAL PORT

The Serial API allows the use of a serial port in the Nokia 12 module (see Figure 3). The serial port is used through the connector API.

The `read()` method is blocking and polling the received data buffer takes a great deal of processor time.

The address syntax of the connection is as follows:

```
StreamConnection sc;
sc = (StreamConnection)Connector.open("comm:3;baudrate=115200");
```

Acceptable baud rates are 1 200,2 400, 4 800, 9 600, 19 200, 28 800, 38 400, 57 600, and 115 200. The default baud rate is 19 200. Baud rate is the only configurable connection parameter. Other settings are fixed: databits = 8, stop bits= 1, and no parity.

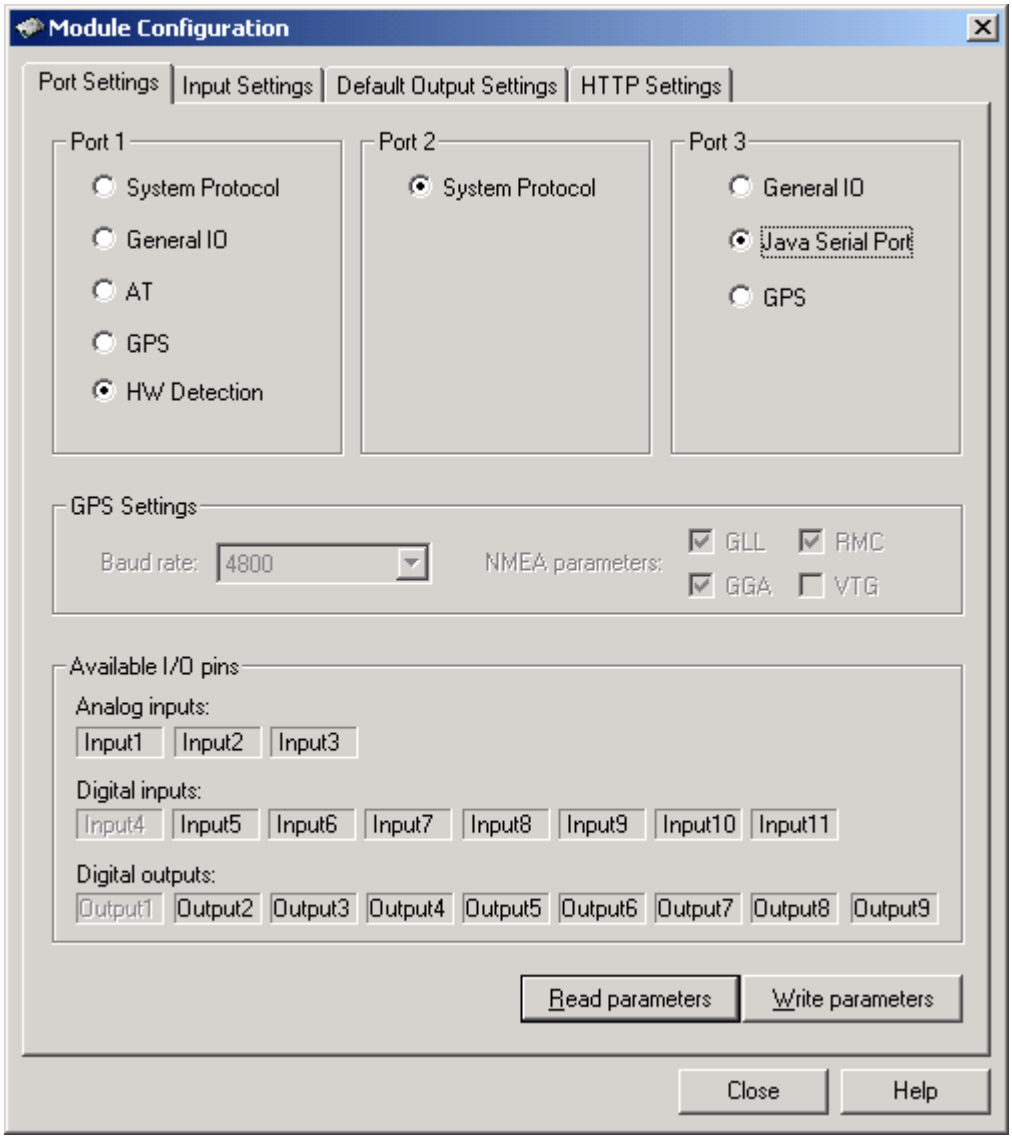

Figure 3. Nokia 12 Configurator settings for using the Java serial port

5.5.1 Serial port example IMlet

This example IMlet works as a simple echo in the serial port:

```
import javax.microedition.midlet.*;
import javax.microedition.midlet.MIDletStateChangeException;

import java.io.InputStream;
import java.io.OutputStream;


import javax.microedition.io.Connector;

/**
 * Simple echo IMlet using serial port
```

```
 */

public class AvailableTest extends MIDlet implements Runnable{


   StreamConnection conn=null;
   InputStream is=null;
   OutputStream os= null;
   int  received=0;
   byte[] data;

   protected void pauseApp()
   {
   }

   protected void destroyApp(boolean parm1) throws
                                    MIDletStateChangeException
   {

   }

   protected void startApp() throws MIDletStateChangeException
   {
      new Thread(this).start();
   }

   public void run()
    {

      try{

      /**
      * Open a serial connection
      */
      conn = (StreamConnection)Connector.open("comm:3;baudrate=115200");


      is = conn.openInputStream();

      os=conn.openOutputStream();

      /* infinite loop */
      while(true){

         /**
         * Rotate until something received and print "rotate" to serial on
         * every loop
         */
         while((received=is.available())==0){
            os.write("rotate".getBytes());
            // Sleep a while.
            Thread.sleep(10000);
         }
         /**
          * print how much data is available
          */
         os.write("Received ".getBytes());
         os.write(Integer.toString(received).getBytes());
         // make an new byte array size of received data
         data=new byte[received];
         /**
         * Read received bytes from the serial port
         * to byteArray data from offset 0 and as much as
         * data received.
         */
         is.read(data,0,received);

         /**
         * Print received byteArray back to serial port
```

```
            */
            os.write(data);

            }//end of while(true)
          }catch(Exception e){
          }
        }
}
```

5.6 WIRELESS MESSAGING API

The Wireless Messaging API can be used to send and receive short messages.

The Wireless Messaging API specification is available at (http://jcp.org/aboutJava/communityprocess/final/jsr120/index2.html) and the Javadocs are available at http://java.sun.com/products/wma/.

The application must first obtain an instance of the `MessageConnection` through the `Connector` class. The URL passed to the `javax.microedition.io.Connector.open()` method identifies the protocol to be used (SMS), and the phone number and/or port of the target:

```
MessageConnection conn;
conn = (MessageConnection)Connector.open("sms://+0123xxxx");
```

Valid URLs are as follows:

- sms://+0123xxxx
- sms://+0123xxxx:5678
- sms://:5678

Opening a connection

As in any communication that uses IMlets, the time period from starting the Nokia 12 module to registering into the network can vary in different networks. An IMlet can use the CORBA API to monitor the network registration status before trying to communicate over the wireless bearer. Another way to wait for the network registration is simply to try and open a connection, catch the raised exception, and try again until the connection succeeds.

```
import javax.microedition.io.*;
import javax.wireless.messaging.*;


public class SmsSender {
 MessageConnection conn = null;
 String url = "sms://+0123xxxx";
```

```
 try {
    conn = (MessageConnection)Connector.open( url );
 }
 catch( Exception e ){
 }
 finally {
    if( conn != null ){
        try { conn.close();
        }catch( Exception e ){}
    }
 }
}
```

Sending a message

To send a message, use the `MessageConnection.newMessage()` method to create an empty message, set its payload (text or binary data to be sent), and invoke the `MessageConnection.send()` method as follows:

```
public static void sendTextSM(String phoneNumber,String text)
         {
         //Set the correct target address.
         String targetAddress = "sms://"+phoneNumber;
         TextMessage       msg;
         MessageConnection sendConn = null;
             try
             {
                 // Creating a MessageConnection to the target address.
                 sendConn        =
                     (MessageConnection)Connector.open(targetAddress);
                 // Creating a text message to be send.
                 msg       =
            (TextMessage)sendConn.newMessage(MessageConnection.TEXT MESSAGE
            );
                  //Set the message text.
                 msg.setPayloadText(text);
                 // Send the message.
                 sendConn.send(msg);
             }catch (Exception e){
             }finally{
                 // Close the MessageConnection if it is still open.
               try{
                     sendConn.close();
                  }catch(Exception ex) {}
             }
         }
```

Receiving a message

The server connection must be opened with the port number, but without the phone number. After opening the connection, the `MessageConnection.receive()` method is called. This returns the next available message to the specified port. If there is no message available, the method blocks until a message arrives or a different thread closes the connection.

```
import java.io.*;
import javax.microedition.io.*;
import javax.wireless.messaging.*;

MessageConnection conn = null;
String url = "sms://:4444"; // only port number!

try {
    conn = (MessageConnection) Connector.open( url );
    while( true ){
        Message msg = conn.receive(); // NOTE!, blocks!!!
        if( msg instanceof BinaryMessage ){
            byte[] data =
                 ((BinaryMessage) msg).getPayloadData();

            //  . . . process binary message . . .

        } else {
            String text =
                 ((TextMessage) msg).getPayloadText();

            // . . . process text message . . .

        }
    }
}
catch( Exception e ){
}
finally {
    if( conn != null ){
        try { conn.close(); } catch( Exception e ){}
    }
}
```

Note: The port number sets limitations for receiving short messages. The Wireless Messaging API is able to receive short messages with a port number. The Nokia 12 module has an alternative way to receive short messages without a port number. For more information, please refer to Chapter 6.

5.7 IOCONTROL API

The IOControl API is a Nokia proprietary API. IMlets using this API cannot be used with M2M devices from other manufacturers.

The `IOControl` class is used to control the input and output pins of the Nokia 12 module. The number of available pins depends on the current port settings of the Nokia 12 module. The Nokia 12 Configurator is a useful tool for checking the available pins in the current configuration (see Figure 3).

The `IOControl` class has the following methods:

- `int getAnalogInputPin(int pin)`
  - This method reads the analog value of the analog input pins 1 - 3.

- `int getAnalogOutputPin(int pin)`
  - This method returns the current analog value to the wanted analog pin. **Note!** This method is not supported in the Nokia 12 module.
- `boolean getDigitalInputPin(int pin)`
  - This method reads the digital value of the input pins 4 - 11.
- `boolean getDigitalOutputPin(int pin)`
  - This method reads the digital value of the output pins 1 - 9.
- `IOControl getInstance()`
  - This is Singleton type method that returns only one instance.
- `void setAnalogOutputPin(int pin, int value)`
  - This method sets the wanted analog value to the wanted analog pin. **Note!** This method is not supported in the Nokia 12 module.
- `void setDigitalOutputPin(int pin, boolean state)`
  - This method activates and deactivates the output pins 1 - 9.

The IOControl API is thread-safe.

	**Note:** When the output state is modified using the IOControl Java API, the state is not stored in the persistent memory and is thus lost when the Nokia 12 module restarts. If the IOControl CORBA API is used to set the I/O pin state, the state is stored in the persistent memory and restored when the Nokia 12 module restarts. Even if the Nokia 12 module remembers the output pin state and sets the pin to the last known state after it restarts, the pin may be in an unstable state during start-up. This must be taken into account in system design.

5.7.1 IOControl example IMlet

This simple IOControl IMlet can switch all available pins on and off. If the pin throws an exception, the exception is ignored.

```
import javax.microedition.midlet.*;
import javax.microedition.midlet.MIDletStateChangeException;
// Import the IOControl class.
import com.nokia.m2m.imp.iocontrol.*;

public class IOControlExample extends MIDlet implements Runnable{

   IOControl ioc = null;


   protected void pauseApp()
   {
   }
```

```
   protected void destroyApp(boolean parm1) throws
                                MIDletStateChangeException
   {

   }

   protected void startApp() throws MIDletStateChangeException
   {
      // Start the thread.

      new Thread(this).start();
   }

   public void run()
    {
      // use the getInstance() method to receive an iocontrol object.
      ioc= IOControl.getInstance();


      // Set all the pins off.
      for(int i=1;i<10;i++) {
         try {
               ioc.setDigitalOutputPin(i,false);
         } catch ( Exception ignored ){}
      }

      // Rotate and rotate......
      while(true){
            for(int i = 1; i < 10; i++){

               // Read the pin value and set it to an opposite value.
               // Catch possible erroneous values in pin numbers.
               try{


               // Sleep for a while to make
               // the pin state change visible in test board.
               Thread.sleep(200);

               // Read the pin value and set it to an opposite value.
               // Catch possible erroneous values in pin numbers.
               ioc.setDigitalOutputPin(i, !ioc.getDigitalOutputPin(i));

               }catch(Exception ignored){}
            }
      }
   }
}
```

5.8 WATCHDOG API

The `WatchdogTimer` class can be used to reboot the Nokia 12 module if the Java VM or IMlet fails to execute. When set, the watchdog timer starts to run towards zero from the set timeout value. When zero is reached, the Nokia 12 module will reset.

The watchdog timer must be placed in the application's main loop. This way it protects against Java VM and IMlet failures such as program deadlock.

The `WatchdogTimer` class includes the following methods:

- setTimeout(int seconds)

  - This method starts the Watchdog timer and throws the IllegalArgumentException.
  - Value 0 (zero) is used to disable the timer.
  - Values from 60 up to 86 400 are used to set the timer.
- resetTimer()
  - This method restarts the counter from a value previously set by the `setTimeout(int seconds)` method.

The Nokia 12 module has only one Watchdog timer. Although you can create multiple instances of the `WatchdogTimer` class, only one native timer is running.

When exiting the IMlet, the watchdog timer should be removed with the `setTimeout(0)` method call. Exiting the IMlet does not stop the watchdog timer, so the Nokia 12 module will reset if the timer was not stopped before exit.

Because Java is running on low priority and the execution time of the IMlet can vary a great deal, timing the watchdog timer can be difficult. Connections and other activities of the Nokia 12 module affect the execution time of the IMlet. A common practice is to use three times the time needed for execution without any disturbance as the watchdog timer interval.

In most M2M applications, the watchdog timer should always be running. It ensures that the system returns to normal function even if the IMlet crashes or the IMlet code deadlocks because of non-obvious reasons in the development phase.

The timeout interval must be set high enough so that IMlet update during the timeout interval is possible. The timeout interval can be set, for example, to 30 minutes and the update interval to 10 seconds. This gives almost 30 minutes of runtime after the IMlet has been shut down for the update. If the IMlet is updated remotely, the update procedure first stops the IMlet and loads a new IMlet Suite, and then restarts the IMlet. 30 minutes is enough to complete the remote update without the Nokia 12 module reset interrupting the update procedure.

Correct use of the watchdog timer requires that it be set using a feasible timeout value and refreshed at correct locations in the Java code. Usually, it is not enough to use the watchdog timer to make sure that Java VM and IMlet keep running. Location of the watchdog timer in the code should be such that it ensures that the Nokia 12 module restarts if the IMlet fails to do the tasks it was designed to do.

5.8.1 Watchdog timer example IMlet

This example IMlet demonstrates the use of the watchdog timer. The IMlet reads the analog input pin value and writes the value to the RMS storage. The

watchdog timer is used to make sure that the IMlet keeps running. A timeout value of 10 minutes is used to allow the IMlet to be updated remotely. The example IMlet uses an `RMSLogger` example class described in Chapter 8.

```
public class WatchdogIMlet extends MIDlet {

   private Log rmsLogger;

   public void startApp() throws MIDletStateChangeException {

      final int pin = 1;
      int value;

      logSerial("WatchdogIMlet starting");


      // Start the timer.
      WatchdogTimer wd = new WatchdogTimer();
      wd.setTimeout(10 * 60); // 10 minutes

      try {

         // Initalize the RMSLogger.
         ORB orb = ORB.init(null, null);
         rmsLogger = RMSLogger.getInstance(orb);

         log("WatchdogIMlet started");

         // Read the input pin value every 60 seconds.
         while (true) {

            value = IOControl.getInstance().getAnalogInputPin(pin);
            log("port " + pin + " value " + value);

            // Reset the timer after every successful reading.
            wd.resetTimer();
            // Wait one second.
            Thread.sleep(60000);

         }

      } catch (Exception ignored) {
         // The above code does not throw exceptions.
         // Exit the loop. The watchdog timer resets the Nokia 12
         // module after 10 minutes.
         log(ignored.toString());
      }

   }

   /**
    * Write log message
    * @param msg
    */
   private void log(String msg) {

      // Write the message also to the serial port.
      logSerial(msg);
      try {
         logRms(msg);
      } catch (Exception e) {
         logSerial(e.toString());
      }
   }

   /**
```

```
     * Write log to the RMS file.
     * @param msg
     */
    private void logRms(String msg) {
       rmsLogger.write(msg);
    }

    /**
     * Write a log message to the serial port.
     * @param msg
     */
    private void logSerial(String msg) {
       SerialPortLogger.getInstance().write(msg);
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean arg0) throws MIDletStateChangeException {
    }

}
```

6. CORBA PROGRAMMING WITH THE NOKIA 12 J2ME™ ORB

CORBA provides the possibility for object-oriented, distributed programming. It is an independent programming language and provides a standardized, application level, remote procedure call framework that is not dependent on programming languages, operating systems, or network protocols.

CORBA allows services to be programmed with tools and programming languages best suited for each device. The main target of using CORBA is to separate networks and communication protocols from application level logic using CORBA as middleware.

The Nokia12 module extends this architecture to wireless networks. CORBA services located in the AM can be written, for example, in C and C++ languages. Services located in the IMlet are written in Java language, and in the customer server application with any programming language that supports CORBA including, but not limited to, C, C++, and Java languages. All these services can communicate with each other over various bearers supported by the Nokia M2M Platform. The Interface Definition Language (IDL) is used to define service interfaces between communicating parties.

Java VM in the Nokia 12 module includes a CORBA ORB API that IMlets can use to call remote CORBA services and implement these services in the IMlet. The ORB API is designed to follow CORBA Java language mapping rules. This way, developers familiar with CORBA and J2ME ORB implementations can quickly start using CORBA in their IMlets.

Even though CORBA is not needed in the end-user application, it may be useful to read this chapter because all set-up and monitoring services provided by the Nokia 12 module are used through the ORB API. Knowledge of how to use the J2ME ORB API to call Module ORB services from the IMlet is needed at least to seize all capabilities provided by the Nokia 12 module.

For more information on CORBA knowledge, please refer to CORBA literature.

6.1 MINIMUM CORBA IMPLEMENTATION

In a device like the Nokia 12 module there are limitations mainly in the amount of available memory. Because of this it is not possible to implement the full CORBA 2.3 ORB. A subset called Minimum CORBA, designed for systems that have limited resources, is used instead. For details, please refer to the Object Management Group's (OMG's) Minimum CORBA specification available at http://www.omg.org/cgi-bin/doc?formal/02-08-01.

Minimum CORBA contains the basic functionality of CORBA, but large parts have been omitted or reduced.

Omitted parts:

- Dynamic Invocation Interface
- Dynamic Skeleton Interface
- Dynamic Anys
- Most parts of the Interface Repository
- Support for setting a default servant for the Portable Object Adapter (POA)
- AdapterActivator and ServantManagers. Because of the omission of ServantManagers, the derived ServantActivator and ServantLocator interfaces are also omitted.
- DefaultServant that has been omitted because of the fixed value of the RequestProcessingPolicy.

Reduced parts:

- Configurable POA policies reduced from the original seven to the following three:
  - LifeSpanPolicy
  - ObjectIdUniquenessPolicy
  - IdAssignmentPolicy.
  - The other four policies have fixed values and cannot be changed: ThreadPolicy(ORB_CTRL_MODEL), ServantRetentionPolicy(RETAIN), RequestProcessingPolicy(USE_ACTIVE_OBJECT_MAP_ONLY), and ImplicitActivationPolicy(NO_IMPLICIT_ACTIVATION).
- Some limitations to full IDL because of the CLDC profile. IDL does not contain the following Java data types that are thus not supported:
  - double
  - float
  - java.math.BigDecimal
- Because CLDC does not contain the `java.util.Properties` class, ORB initialization has been changed from `ORB.init(String[] args, java.util.Properties props)` to `ORB.init(String[] args, java.util.Hashtable props).`

Some features have been added to Minimum CORBA to make application programming simpler.

Added parts:

- Support for portable stubs and skeletons. This helps to generate codes for stubs and skeletons using Java IDL compilers provided that they can generate CLDC-compatible code.
- Support for several POAs.

6.1.1 Designing and implementing a typical CORBA application

This chapter gives a top-level procedure for making a CORBA application.

To design and implement a CORBA application:

1. Design the interface definition. Notice the data type limitations mentioned in Chapter 6.1. The IDL defines the services used/implemented by CORBA applications. It is an interface agreement between the CORBA client and server.
2. Compile the IDL as described in Chapter 6.1.2.
3. Modify the results of IDL compilation as described in Chapter 6.1.2. This step is needed because of the CLDC profile unless you have a CLDC-compatible IDL compiler.
4. Implement the servants and possible client applications as described in Chapter 6.2.3
5. Implement the IMlet as described in Chapter 6.2.3.This application initialises the ORB and sets up the ORB and servants.

6.1.2 IDL compiler

IDL is used to define the API provided by the server for remote clients. Minimum CORBA supports the full IDL excluding some features that have been omitted because of the limitations of the CLDC 1.0 profile (see Chapter 6.1).

6.1.2.1 Compiling an example IDL

An example IDL contained in a file called hello.idl

```
module hello
{
   interface Greeter
   {
      string sayHello(in string name);
   };
};
```

This is an IDL of a simple program that has a string as the parameter in the remote call and also returns a string.

The following command-line example shows how to compile the example IDL using Sun's IDL compiler from J2SE SDK 1.4.2.

```
idlj -fallTie hello.idl
```

After the compilation procedure is finished, the following classes are contained in the `hello` package: `_GreeterStub`, `GreeterHelper`, `GreeterHolder`, `GreeterOperations`, `GreeterPOA`, `GreeterPOATie`.

From these classes, `GreeterPOA` and `GreeterPOATie` belong to the server side and are known as skeleton classes in CORBA terminology. The client program does not need them. The `_GreeterStub` class is known as stub class in CORBA terminology and the CORBA client needs it. However, when you implement both the server and client in the Nokia 12 module, all the classes are needed in the same JAR file as the server application. The `GreeterPOATie` class is not needed if the implementation does not use it.

Two approaches are available in the J2ME ORB for servant implementation. The first approach is to inherit the servant implementation class from the xxxxPOA class generated by the IDL compiler. This approach is used in the example servants described in this document. The second approach is to use a tie pattern with generated xxxxPOATie classes. Using the tie pattern leaves an open possibility to inherit the servant implementation class from some other class.

Tip: The JacORB IDL compiler supports generating CLDC 1.0 compatible stubs and skeletons that can be used with Nokia 12 module without the modifications explained in chapter 6.1.2.2. Use the `-cldc10` command line option to generate J2ME-compatible stubs and skeletons. The JacORB is available from the http://www.jacorb.org/.

6.1.2.2 Modifying classes generated by the Sun IDL compiler

If you are using Sun's IDL compiler from the Java SDK, this chapter instructs how to modify the generated code. If your compiler generates CLDC 1.0 - compatible code, the instructions given in this chapter are not needed.

Sun's IDL-to-Java compiler produces code that is not J2ME-compatible. The errors are repeated in multiple places depending on the type and size of your IDL file. The simplest way to detect incompatibilities is to compile classes using the **classes.zip** file from the Nokia 12 Concept simulator as the system classes with the development tool, and to pre-verify the code using a pre-verification tool (**preverify.exe**) from Sun's Wireless Toolkit. If neither the compilation nor

the pre-verification phase report errors, the code should not contain any incompatible code. For more information, please refer to Chapter 4.8.

Tip: Using an IDE is recommended for this task.

The Stub and Helper classes create the most problems during compilation.

Modifying the generated Stub classes

Stubs require changes to the `readObject()`, `writeObject()`, and `_IDS` methods.

The IDL compiler generated a class called `_GreeterStub.java` from the **greeter.idl** file (see Chapter 6.1.2.1). This class contains two methods that are not J2ME-compatible:

- readObject()
- writeObject()

Because both of these methods are used with the Remote Method Invocation (RMI) that is not supported by J2ME, the methods are not needed and can be deleted from the Stub class.

The Stubs also contain a method called `_ids()` that returns a string array, for example:

```
public String[] _ids ()
{
   return (String[])  ids.clone ();
}
```

Because the `clone()` method is not supported by CLDC, it must be changed to:

```
public String[]  ids ()
{
   return __ids;
}
```

Modifying the generated Helper classes

For each interface you have defined in your IDL, the IDL compiler produces a `Helper`. The IDL compiler generated a class called `GreeterHelper.java` from

the **greeter.idl** file (see Chapter 6.1.2.1). This class contains two methods that are J2ME-compatible:

- read()
- type()

The `read()` method has the following structure:

```
public static hello.Greeter read()
{
   return narrow (istream.read_Object (_GreeterStub.class));
}
```

Since the static field `class` is not included in the CLDC, the code has to be modified. Thus, you have to remove the `_GreeterStub.class` parameter from the invocation of the `read_Object()` method. Modify the method as follows:

```
pub public static hello.Greeter read()
{
   return narrow(istream.read Object ());
}
```

Helpers may also have incompatible code in their `type()` methods that can access the static variable class that is not contained in the CLDC.

When the `idlj` command is used for generating the code, the `type()` method can be already CLDC-compatible, but there may also be situations where the `type()` method contains the following code:

```
private static org.omg.CORBA.TypeCode   typeCode = null;
private static boolean __active = false;
synchronized public static org.omg.CORBA.TypeCode type ()
{
   if (  typeCode == null)
   {
      synchronized (org.omg.CORBA.TypeCode.class)
      {
         if (__typeCode == null)
         {
               .
               .
               .
         }
      }
   }
   return   typeCode;
}
```

Remove the code that is synchronised to the static variable class. You only need to remove the line and the associated brackets where the synchronising is done. Also, remove the first line and the associated brackets where it is checked whether the `__typecode` is null. There is no need to check it twice. The method should appear as follows:

```
private static org.omg.CORBA.TypeCode __typeCode = null;
private static boolean   active = false;
synchronized public static org.omg.CORBA.TypeCode type ()
{
   if (  typeCode == null)
   {
         .
         .
         .
   }
   return   typeCode;
}
```

Modifying the generated POA Skeleton classes

The Skeleton classes contain an `_all_interfaces()` method that needs to be modified.

Example code:

```
public String[] _all_interfaces (org.omg.PortableServer.POA poa,
                                 byte[] objectId)
{
   return String[])  ids.clone ();
}
```

Change the example code to:

```
public String[] _all_interfaces (org.omg.PortableServer.POA poa,
                                 byte[] objectId)
{
   return   ids;
}
```

6.1.2.3 Design issues

It is important to keep in mind that resources are limited in the J2ME environment. The main concern with IDL is the amount of classes and hence the size of the application JAR file. The IDL should be kept as simple as possible because, for example, every interface produces a number of classes and all type definitions from the IDL map to at least one class in the Java code.

If the interface definition is extensive and the IMlet needs or uses only a part of it, the unnecessary parts can be reduced and recompiled for the IMlet. An alternative solution is to include only the classes (produced by the IDL compiler) needed by the IMlet into the application JAR file.

Also, CORBA servers implemented into the IMlet should be designed to be stateless if possible. You must be prepared for a connection failure that may occur in the middle of a remote invocation. In that case, the server should not crash or be left in a state where it can no longer function properly.

6.1.3 Connection issues

The J2ME ORB uses TCP connections to communicate with other ORBs. Wireless bearers supported by the Nokia 12 module carry TCP. CSD/HSCSD and GPRS/EGPRS bearers are used when communicating over the wireless link and M2M System Protocol is used when communicating with the AM. Local loop-back sockets are used when communicating with the Module ORB. For more information on communications and addressing, please refer to Chapter 3.1.

By default, the J2ME ORB listens to incoming TCP connections from port 19760. You must not create a TCP server socket into this port using the Java socket API in IMlets that are using CORBA. This will prevent the J2ME ORB from starting. The port can be changed by a configuration parameter, but because it is used as the default port throughout the documentation, the change is not recommended. The ORB starts to listen to incoming connections immediately after it is initialised and closes the server socket when the ORB is destroyed.

When the ORB is initialised and is listening for incoming connections, it does not start a wireless link. The wireless link is initialised automatically when there is outgoing traffic, that is CORBA requests, going to addresses that are not part of the wireless network. When other ORBs want to communicate with the J2ME ORB, they must first establish a network connection. The J2ME ORB automatically starts to use the wireless network link opened by another service.

When the CSD bearer is used and the last TCP socket is closed, the J2ME ORB automatically closes the link. With the GPRS bearer, the J2ME ORB functions differently. It does not automatically close the GPRS network link; even if the last TCP connection over GPRS is closed, the GPRS context stays open. If shutdown of the GPRS network link is required, you can set the connection timeout property for the bearer using the Nokia 12 Configurator. When the connection timeout is set above 0, the Nokia 12 module closes the wireless network link after detecting that no data has been sent or received over the link within the defined time period.

The J2ME ORB needs the IP address negotiated for the network link to publish object references. To meet this requirement the network link must be open when the J2ME ORB creates the object reference. This generates a limitation in

which object reference publishing is only possible in the invocation context. The invocation context refers to a situation in which a servant located in an IMlet is processing a server-originated request.

Although the J2ME ORB and Nokia 12 module handle communication automatically, the J2ME ORB provides interfaces for selecting the used wireless bearer from those configured into the Nokia 12 module. With the help of the Module ORB services, an IMlet can even manage bearer configuration. That is, the IMlet can programmatically set all the needed settings.

6.2 EXAMPLE CORBA APPLICATION

This chapter presents typical 'Hello world' examples. One example has the IMlet acting as a server and the J2SE application acting as a client. The other example has the IMlet acting as a client and the J2SE application acting as a server.

Figure 4 presents a typical example of a client IMlet using the services of a J2SE server.

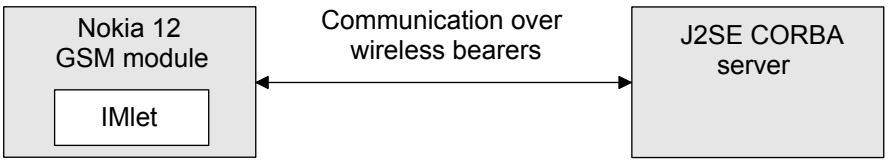

Figure 4. IMlet that uses the services of a J2SE server

6.2.1 ORB initialization and shutdown

There can be only one J2ME ORB instance in the Nokia 12 module.

The following code initializes the ORB into the Internet Inter-ORB Protocol (IIOP) mode:

```
Hashtable properties = new Hashtable();
properties.put("com.nokia.m2m.orb.UseM2MGateway", "no");
orb = ORB.init(null, properties);
```

Subsequent calls to the `ORB.init(String[] args, Hashtable properties)` result in a java.io.IOException. You should save the reference retrieved from the first and only call. The `ORB.destroy()` method must be called before the ORB can be re-initialized.

Calling the `ORB.init()` method that has no parameters always returns the same reference to a singleton ORB that cannot be used for making or accepting remote invocations. It can be used to create `Any` types, for example.

The ORB is set to IIOP mode by giving a 'no' initialisation parameter to the 'com.nokia.m2m.orb.UseM2MGateway'. In this mode the ORB communicates directly with server application using standard IIOP communication rules.

The default mode in the Nokia 12 module is 'M2M mode' for legacy support reasons. The mode of the ORB has an effect on communications and object reference publishing. If the ORB is left to the 'M2M mode', all wireless communications go to the gateway address specified in the wireless bearer settings.

6.2.2 Simple client program HelloWorldClient

This chapter describes how to create simple example server and client programs using the classes produced in Chapter 6.1.2.

The J2SE application functions as the server and IMlet as the client.

6.2.2.1 Creating a J2SE™ server

To create a J2SE server:

1. Implement a servant and insert it into the `hello.impl` package.

```
package hello.impl;

import hello.GreeterPOA;

public class GreeterImpl extends GreeterPOA
{
   // This is the method that is invoked by the remote call.
   public String sayHello(String name)
   {
      return "Hello world and " + name + "!";
   }
}
```

2. Create a J2SE `HelloServer`.

	**Note:** This code runs in the server application. You cannot compile the following code into an IMlet.

```
package hello.impl;

import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.CosNaming.*;

public class HelloServer
```

```
{
   private ORB orb = null;

   public HelloServer()
   {

   }

   public static void main(String[] args)
   {
      HelloServer server = new HelloServer();
      server.runServerFunc(args);
   }

   private void runServerFunc(String[] args)
   {
      try
      {
         // Initialise the J2SE ORB.
         orb = ORB.init(args, System.getProperties());

         // Resolve the RootPOA.
         POA rootPoa =
            POAHelper.narrow(orb.resolve_initial_references("RootPOA"));

         // Create policies for a persistent POA.
         org.omg.CORBA.Policy[] persistentPolicies =
            new org.omg.CORBA.Policy[] {
               rootPoa.create_id_assignment_policy(
                  IdAssignmentPolicyValue.USER_ID),
               rootPoa.create id uniqueness policy(
                  IdUniquenessPolicyValue.UNIQUE ID),
               rootPoa.create lifespan policy(
                  LifespanPolicyValue.PERSISTENT),
               rootPoa.create_servant_retention_policy(
                  ServantRetentionPolicyValue.RETAIN)};

         // Create the persistent ServerPOA.
         POA serverPoa =
            rootPoa.create POA(
               "ServerPOA",
               rootPoa.the_POAManager(),
               persistentPolicies);

         // Create an implementation of the servant.
         GreeterImpl impl = new GreeterImpl();

         // Activate the servant and retrieve a reference to it.
         serverPoa.activate object with id("HelloServer".getBytes(), impl);
         org.omg.CORBA.Object greeterRef =
            serverPoa.id to reference("HelloServer".getBytes());

         // Activate the POAManager. It is a very common mistake to forget
         // this. No requests will be processed for servants that do not
         // have their POA activated.
         rootPoa.the POAManager().activate();

         // Resolve the Naming Service.
         org.omg.CORBA.Object nsRef =
         orb.resolve initial references("NameService");
         NamingContextExt ns = NamingContextExtHelper.narrow(nsRef);

         // Publish the reference with a name "HelloServer".
         // This name can be used to resolve an IOR to the servant.
         ns.rebind(ns.to_name("HelloServer"), greeterRef);

         // This call keeps the application running.
         orb.run();
      }
```

```
      catch (Exception e)
      {
         e.printStackTrace();
      }
      finally
      {
         // Destroy the orb if it was initialized.
         if (orb != null)
         {
            orb.destroy();
         }
      }
   }
}
```

6.2.2.2 Creating a client IMlet

A bearer must be configured to the Nokia 12 module for the client IMlet to work.

The client IMlet uses a CORBA Naming Service to resolve the target server application. The CORBA Naming Service address is hard-coded into the IMlet.

```
package com.nokia.m2m.sdk.examples.hello.impl;

import java.util.Hashtable;

import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;
import org.omg.CORBA.ORB;
import org.omg.CosNaming.*;
import com.nokia.m2m.sdk.examples.hello.idl.hello.*;
import com.nokia.m2m.sdk.examples.logger.SerialPortLogger;
;

/**
 * Client IMlet for Hello World application.
 *
 *
 */
public class HelloClientImlet extends MIDlet {

   // Address of CORBA Name Service used resolve server location.
   private String NAMING_SERVICE_ADDRESS = "10.35.1.195:900";

   private ORB orb = null;
   public HelloClientImlet() {
      super();
   }

   protected void startApp() throws MIDletStateChangeException {
      try {

         log("HelloClientImlet starting");

         log("waiting ...");
         // wait to get into GSM network service
         Thread.sleep(30);
         log("continue");

         // Initialize the J2ME ORB in IIOP mode.
```

```
        Hashtable properties = new Hashtable();
        properties.put("com.nokia.m2m.orb.UseM2MGateway", "no");
        orb = ORB.init(null, properties);

        // Object URL is in format
        //  "corbaloc::10.35.1.195:900/NameService"
        String nsURL = "corbaloc::" + NAMING_SERVICE_ADDRESS +
              "/NameService";
        org.omg.CORBA.Object nsRef = orb.string_to_object(nsURL);

        log("Resolving Naming Service");
        NamingContextExt nc = NamingContextExtHelper.narrow(nsRef);
        log("Resolving server");

        // Resolve "HelloServer" from Naming Service
        NameComponent[] nca = new NameComponent[] { new
                        NameComponent("HelloServer", "")};

        log("Resolving HelloServer");
        // Throws exception if fails
        org.omg.CORBA.Object greeterRef = nc.resolve(nca);
        log("HelloServer found");
        // Now, narrow the Reference
        log("Narrowing HelloServer");
        Greeter greeter = GreeterHelper.narrow(greeterRef);

        // Make the invocation.
        log("Invoking sayHello()");
        log(greeter.sayHello("IMlet-client"));

      } catch (Exception e) {
        log(e.toString());
      } finally {
        // Remember to destroy the ORB
        if (orb != null) {
          orb.destroy();
        }
      }
    }

    protected void pauseApp() {
    }

    protected void destroyApp(boolean arg0) throws
                                        MIDletStateChangeException {
      if (orb != null) {
        orb.destroy();
      }
     }

    private void log(String msg) {
      SerialPortLogger.getInstance().write(msg);
    }
}
```

The client is ready to be started. The client JAR file contains the `_GreeterStub`, `GreeterHelper`, `GreeterHolder`, and `GreeterOperations` classes in the `hello` package and the `HelloClientImlet` class in the `com.nokia.m2m.sdk.examples.hello.impl` package.

6.2.3 Simple servant program HelloWorldServer

In this example, IMlet functions as the server and J2SE application as the client.

6.2.3.1 Creating a server IMet

To create a server IMlet:

1. Implement the servant and insert it into the `com.nokia.m2m.sdk.examples.hello.impl` package.

```
package hello.impl;

import hello.GreeterPOA;

public class GreeterImpl extends GreeterPOA
{
   // This is the method that is invoked by the remote call.
   public String sayHello(String name)
   {
      return "Hello world and " + name + "!";
   }
}
```

2. Create a server IMlet that instantiates the `GreeterImpl` class and accepts the incoming remote invocations.

```
package hello.impl;

import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;
import com.nokia.m2m.sdk.examples.logger.SerialPortLogger;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;

public class HelloServerImlet extends MIDlet
{
   private ORB orb = null;

   /**
    * Constructor for HelloServer.
    */
   public HelloServerImlet(){
   }

   protected void startApp() throws MIDletStateChangeException
   {
      try
      {
         // Initialize the J2ME ORB in IIOP mode.
         Hashtable properties = new Hashtable();
         properties.put("com.nokia.m2m.orb.UseM2MGateway", "no");
         orb = ORB.init(null, properties);

         // Get the RootPOA.
         POA rootPoa =
            POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
```

```
        // Create policies for a persistent POA.
        org.omg.CORBA.Policy[] persistentPolicies =
           new org.omg.CORBA.Policy[] {
              rootPoa.create id assignment policy(
                 IdAssignmentPolicyValue.USER_ID),
              rootPoa.create_id_uniqueness_policy(
                 IdUniquenessPolicyValue.UNIQUE ID),
              rootPoa.create lifespan policy(
                 LifespanPolicyValue.PERSISTENT)};

        // Create the server POA. Retrieve the POAManager from the
        // RootPOA and use the persistent policies.

        POA serverPOA = rootPoa.create POA("ServerPOA",
           rootPoa.the POAManager(), persistentPolicies);


        GreeterImpl impl = new GreeterImpl();

        // Activate the servant. The objectKey will be
        // ServerPOA/HelloServer.
        serverPOA.activate object with id("HelloServer".getBytes(), impl);

        // Activate the POAManager.
        // No requests will be processed if the POAManager of the
        // POA has not been activated.
        rootPoa.the POAManager().activate();

        // There is no need to call the orb.run() method.
     }
     catch (Exception e)
     {
        log(e.toString());
     }
     finally
     {

        if(orb != null)
        {
           orb.destroy();
        }
     }
  }

  protected void pauseApp()
  {
  }

  protected void destroyApp(boolean arg0) throws
     MIDletStateChangeException
  {
     if (orb != null)
     {
        orb.destroy();
     }
  }

  private void log(String msg)
  {
     SerialPortLogger.getInstance().write(msg);
  }
}
```

The server is now ready to be started. The server application JAR file contains the `_GreeterStub`, `GreeterHelper`, `GreeterHolder`, `GreeterOperations`, and `GreeterPOA` classes in the `hello` package and the `GreeterImpl` and `HelloServerImlet` classes in the `hello.impl` package.

Starting the IMlet does not open a connection, but the server is ready to accept an incoming connection. The server does not publish a reference anywhere, but it always activates its servant with the same object ID. For the client to be able to make an invocation to the server, it must know the IP address of the Nokia 12 module in question.

6.2.3.2 Creating a J2SE client

This J2SE client application uses a corbaloc object URL to resolve the target servant.

```
public class HelloClientJ2SE {

   public static void main(String[] args) {

      try {

         ORB orb =
            org.omg.CORBA.ORB.init((String[]) null,
                                   System.getProperties());

         String objectKey = "ServerPOA/HelloServer";

         // Resolve HelloServer running in Nokia 12 module.
         // Module IP address must be known.
         Object ref =
            orb.string_to_object("corbaloc::10.35.3.4:19760/" + objectKey);

         System.out.println("Narrowing...");
         HelloServer hs = HelloServerHelper.narrow(ref);

         // Remote method invocation
         String ret = hs.hello("hello");
         System.out.println("Server says" + ret);

      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

6.3 THE NOKIA 12 J2ME ORB

This chapter list the classes and most important methods of the ORB API that are used directly and not via the OMG API. This chapter also describes the configuration possibilities of the J2ME ORB.

For more information on classes and methods, please refer to Java documentation.

6.3.1 Configuration values

This chapter contains the possible configuration values used to control the behaviour of the J2ME ORB. The following values can be configured in the J2ME ORB initialization: `RequestTimeout`, `ListenPort`, `GiopVersion`, `GiopVersionOverride`, `ImplName`, `UseM2MGateway`, `ListenIp`, and `UseCodeSetContext`.

Each configuration key has to be prefixed with a header 'com.nokia.m2m.orb.' when given in the J2ME ORB initialisation.

RequestTimeout

The `RequestTimeout` parameter value indicates the time in seconds a reply is waited for before closing the connection. If the time is set to '0' (zero), the J2ME ORB waits until infinity. If the time is set and the limit is reached, a NO_RESPONSE exception is thrown. This parameter can be used to protect the IMlet against losing a General Inter-ORB Protocol (GIOP) reply in communications. The default value for the `RequestTimeout` is '0'.

GiopVersion

The `GiopVersion` parameter value indicates the GIOP version that the J2ME ORB uses when creating and publishing object references. Possible values are '1.0', '1.1', and '1.2'. The default value for the `GiopVersion` is '1.2'.

GiopVersionOverride

The `GiopVersionOverride` parameter value defines whether the J2ME ORB is forced to make invocations using the specified GIOP version. Possible values are '1.0', '1.1', and '1.2'. The `GiopVersionOverride` has no default value set and in that case a GIOP version specified in the IOR is used.

ImplName

The `ImplName` parameter value indicates the J2ME ORB name and it is a part of the object key. A possible value is a string. The `ImplName` is not set by default. For more information, please refer to Chapter 6.3.4.9.

ListenPort

The `ListenPort` parameter value sets the port that the ORB listens to. Possible values range between 0 - 65535. If the value is set to '0' (zero), the Nokia 12 module assigns a free port number. The default value for the `ListenPort` is '19760'.

ListenIp

The `ListenIp` parameter value, together with the `ListenPort` parameter value, is published in the IOR when the ORB is in the IIOP mode. The default value for the `ListenIp` is '0.0.0.0', and the Nokia 12 module uses the negotiated IP address for the wireless link.

UseM2MGateway

The `UseM2MGateway` parameter value defines whether a gateway is used. The possible values are 'yes' and 'no'. The default value for the `UseM2MGateway` is 'yes'. This value must be set to 'no' to enable direct communication with a third party IIOP-compatible ORB.

UseCodeSetContext

The `UseCodeSetContext` parameter value defines whether to add service context information to requests. The default value for the `UseCodeSetContext` is 'yes'. The code set context supports wide character types, wstring and wchar, but it also generates bigger messages.

6.3.1.1 M2M and IIOP modes in J2ME ORB

By default, the J2ME ORB is in the M2M mode because of legacy reasons. In this mode, all wireless communications originating from the Nokia 12 module go to the gateway specified in wireless bearer settings. However, the IIOP mode is recommended for the current applications. The mode can be changed with the `UseM2MGateway` configuration value.

When the IIOP mode is set, the normal IIOP profile is used with IORs. In this mode, the Nokia 12 module can, for example, connect to an office intranet and communicate with CORBA servers located there without the Nokia M2M Gateway software. When the J2ME ORB is set to IIOP mode, the Module ORB should also be set to IIOP mode.

Note: When the J2ME ORB is set to IIOP mode, the Module ORB must also be set to IIOP mode.

6.3.1.2 Initialization example

This chapter presents an example where the J2ME ORB is initialised to use the GIOP version 1.1 and IIOP mode.

```
java.util.Hashtable props = new java.util.Hashtable();

props.put("com.nokia.m2m.orb.UseM2MGateway","no");
props.put("com.nokia.m2m.orb.GiopVersion","1.1");

// Initialise the ORB.
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(null, props);
```

6.3.2 Client programming

In this chapter, the `ORB.string_to_object(String objectURL)` method is used to create a CORBA object reference.

6.3.2.1 corbaloc object URL

When the object URL 'corbaloc::10.35.1.201:900/HelloServer' is defined, it can be parsed into the following elements:

- IP address: 10.35.1.201
- Port: 900
- Object.key: `HelloServer`

This means that the reference points to a servant with the `HelloServer` object key. The servant resides in an ORB that waits for invocations at an IP address 10.35.1.201 and port 900. Only IP addresses in dotted decimal format are allowed here. DNS names cannot be used in object URL strings.

As always with CORBA object references, having a reference does not guarantee that the referenced object actually exists. Communication between a client and a server is initiated during the narrowing or method call phase.

6.3.2.2 Method calls over the wireless link

Whenever the reference points to an address that is not local, a remote method invocation goes through the wireless link, for example:

```
org.omg.CORBA.Object ref =
orb.string_to_object("corbaloc::127.0.0.1:444/LocalService");
```

Here a reference is made that points to a servant activated in the given Nokia 12 module. If the `narrow()` method is called for this reference, the wireless link is not used.

```
org.omg.CORBA.Object ref =
orb.string to object("corbaloc::10.35.1.207:444/RemoteService");
```

This reference points to an object outside the given Nokia 12 module. When a remote invocation is made to this object, it goes through the wireless link to the

target server. Also, the `ORB.createReference()` method can be used for object reference creation.

6.3.2.3 Connecting to the Module ORB services

The Module ORB runs on the Nokia 12 module and is not a Java application. It listens for activations at port 19740 and contains a number of services, for example, Embedded Terminal, Wireless Device, and IO services. The URLs and object keys for retrieving references to these services are described in more detail in Chapter 7.

This chapter presents an example on how to retrieve a reference to the Embedded Terminal service. References are created using both corbaloc and a reference creation technique specific to the J2ME ORB.

```
// Initialise the ORB.
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(null, null);

String etUrl = "corbaloc::127.0.0.1:19740/ORB/OA/IDL:ET:1.0";

// Make a reference to the Embedded Terminal object.
org.omg.CORBA.Object etRef = orb.string to object(etUrl);

// Make a second reference by using a technique specific to the J2ME ORB.
com.nokia.m2m.orb.ORB norb = (com.nokia.m2m.orb.ORB)orb;
org.omg.CORBA.Object etRef2 = norb.createReference("127.0.0.1", 19740,
"ORB/OA/IDL:ET:1.0".getBytes(), "IDL:ET:1.0", "1.2");

// Narrow the references. Since these are local objects, invocations
// do not go to the wireless link.
com.nokia.m2m.orb.idl.et.ET et =
com.nokia.m2m.orb.idl.et.ETHelper.narrow(etRef);

com.nokia.m2m.orb.idl.et.ET et2 =
com.nokia.m2m.orb.idl.et.ETHelper.narrow(etRef2);
```

6.3.2.4 Connecting the Application Module

Connection to an ORB in the application module is done via a serial port. The AM ORB has a server socket in the Nokia 12 module and it listens to incoming method calls via this socket.

In this example, there is an AM ORB with a server socket listening to port 800. It has a servant with an 'AMORB/Example' object key.

Creating a reference to the servant:

```
// Initialise the orb.
org.omg.CORBA.Object amServantRef =
orb.string to object("corbaloc::127.0.0.1:800/AMORB/Example");
```

6.3.2.5 Local calls

It is possible to make a method call to the servant that is running in the same J2ME ORB. It is recommended to use the servant (instance of the

implementation class) directly using the Java method invoke, and not via the CORBA interface. Otherwise all CORBA method invocations use the loop-back socket of the Nokia 12 module and use more resources than Java method calls.

6.3.2.6 Link control

By default, the J2ME ORB uses the default connection set into the Nokia 12 module to communicate with remote servers. This connection can be set through the Wireless Device service provided by the Module ORB. The J2ME ORB can communicate over IP-based bearers like GPRS or CSD.

Note: When using the J2ME ORB, make sure that the selected bearer is not an SMS bearer. The SMS bearer is a special bearer used with the Wake-Up Service over binary short messages, and it cannot be used in CORBA communications. This has nothing to do with sending and receiving text short messages in IMlet code.

For more information on the Module ORB and its services, please refer to Chapter 7.

6.3.2.7 Binding a connection to a CORBA object reference

The J2ME ORB allows you to bind a certain connection, configured into the Nokia 12 module, to an object reference. When the methods of this object are called, the J2ME ORB uses the selected connection.

An example of connection binding:

```
// Initialise the ORB.
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(null, null);

// Cast the ORB to the Nokia ORB.
com.nokia.m2m.orb.ORB norb = (com.nokia.m2m.orb.ORB) orb;

org.omg.CORBA.Object ref=
orb.string to object("corbaloc::10.35.1.195:3000/HelloServer");

// Bind the reference to use the first connection. Zero-based indexing
// is used here.
norb.bindTransport(ref, new int[]{0});
```

The `bindTransport()` method uses zero-based indexing to refer to the configured connections whereas the Nokia 12 Configurator uses one-based indexing. If multiple connection indexes are given, the J2ME ORB tries to open wireless connections in the order in which they were given in the connection index array. The first successfully created connection is used.

For example, you can configure a GPRS connection as the first connection and CSD as the second connection using the Nokia 12 Configurator. The following example sets a reference to use these connections.

```
norb.bindTransport(ref, new int[]{0,1});
```

If the J2ME ORB is unable to open the GPRS network link, it will automatically try to open the CSD link.

6.3.2.8 Exception catching

If a `SystemException` is received from the remote server during invocation, the minor code contained in the exception is helpful.

```
try
{
   // Initialise the ORB, get a reference to the remote object, narrow it
   // and do the remote invocation.
}
catch(org.omg.CORBA.SystemException se)
{
   // Extract the minorcode.
   int minorCode = se.minor;

   // Examine the minorcode.
}
catch(Exception e)
{
   // Handle the exception.
}
```

For more information on minor codes, please refer to the *Nokia 12 GSM Module GIOP Minor Exception Code Descriptions*.

6.3.3 One-way calls

One-way remote invocations do not have a return value. The client does the invocation but the server does not send a response message. The advantage of this approach is that the procedure is faster when compared to a normal CORBA invocation where a response is expected even if the method is not expected to return anything.

With this kind of invocation, an exception is raised only if an error occurs before the request leaves the Nokia 12 module. After that, the results are unknown.

An example IDL for an interface that contains a one-way method:

```
module example
```

```
{
   interface Service
   {
      oneway void notify(in string situation);
   };
};
```

6.3.4 Servants

Servants are targets for the remote invocation. You must implement the xxxPOA classes that are produced by the IDL compiler and activate them on a server that uses a POA and POAManager. After that, clients that include the client-side classes (also known as stubs) and interfaces can make a remote invocation to that servant through the server ORB. Figure 5 clarifies the relationships.

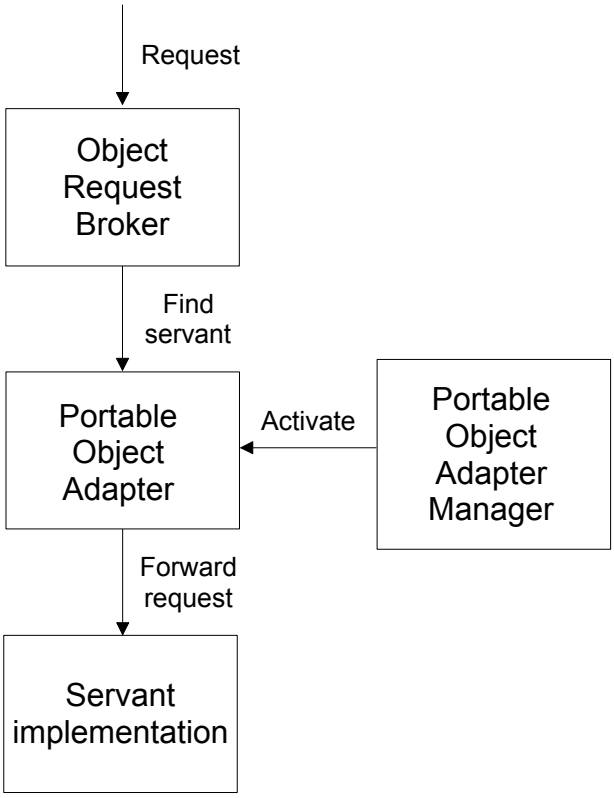

Figure 5. Request processing

6.3.4.1 Portable Object Adapter

POA is used to hold and activate the servants created by the user. For more information on the Minimum CORBA POA implementation, please refer to Chapter 6.1.

6.3.4.2 POA policies

The minimum CORBA POA policies and their effect on POA behaviour are explained in this chapter.

LifespanPolicy

The LifespanPolicy specifies the lifespan of servants activated in a specified POA. When the POA no longer exists, using object references that point to objects under the specified POA result in an OBJECT_NOT_EXIST exception.

The LifespanPolicy can have the following values: TRANSIENT and PERSISTENT. When using the TRANSIENT value, references created from the servants that are activated under a given POA work only as long as the POA exists. Thus the lifetime of the references equals the lifetime of the POA. When using the PERSISTENT value, created references can outlive the POA and the process where they were created. Thus, the references can work even after a whole application has been restarted. This requires that the IMlet initializes the same servant using the same settings every time it starts.

The default value for the LifespanPolicy is TRANSIENT.

ObjectIdUniquenessPolicy

The ObjectUniquenessPolicy defines whether the servants that are activated in a given POA must have unique object identifiers.

The ObjectUniquenessPolicy can have the following values: UNIQUE_ID and MULTIPLE_ID. When using the UNIQUE_ID value, the servant supports only one object ID. Trying to activate a servant that is already active with a different object ID in the given POA results in a SERVANT_ALREADY_ACTIVE exception. When using the MULTIPLE_ID value, the servant can support one or more object IDs.

The default value for the ObjectIdUniquenessPolicy is UNIQUE_ID.

IdAssignmentPolicy

The IdAssignmentPolicy controls whether the object ID of the servant to be activated is generated by the ORB or must be provided by the application.

The IdAssignmentPolicy can have the following values: USER_ID and SYSTEM_ID. When using the USER_ID value, the servants activated with the POA are assigned an object ID provided by the application. When using the SYSTEM_ID value, object ID for the servant to be activated is created by the system.

The default value for the IdAssignmentPolicy is SYSTEM_ID.

6.3.4.3 Transient POA

A transient POA is used for runtime objects and it assigns a unique object ID for the objects activated under it. The object ID is valid only for the lifetime of the servant. Unless otherwise specified, a POA is created with transient policies. The RootPOA is always a transient POA.

The object key that is automatically created for a transient object is unique for the ORB lifetime and can be used to identify the servant. The J2ME ORB does not ensure that object keys for transient objects are unique between the restarts of the J2ME ORB. It is possible that exactly the same key is generated in later ORB sessions if the ORB is shut down and restarted.

Policy values of a transient POA:

- IdAssignmentPolicy: SYSTEM_ID
- IdUniquenessPolicy: UNIQUE_ID
- LifeSpanPolicy: TRANSIENT

6.3.4.4 Persistent POA

A persistent POA is used to hold bootstrap objects. It can be used to assign the same object ID to a servant each time the servant is recreated. Using a persistent POA you can assign a well-known object key for the servant.

Policy values for a persistent POA:

- IdAssignmentPolicy: USER_ID
- IdUniquenessPolicy: UNIQUE_ID
- LifeSpanPolicy: PERSISTENT

6.3.4.5 RootPOA

When an ORB initializes, it creates a RootPOA. POAs can be used to create other POAs, and each created POA is added as a child POA to the creator POA. Thus, the created POAs form a tree and the RootPOA is at the base of that tree. The RootPOA is a transient POA.

The RootPOA is one of the initial references of the ORB and a reference to it can be retrieved as follows:

```
// Initialise the ORB.
Hashtable props = new Hashtable();
props.put("com.nokia.m2m.orb.UseM2MGateway", "no");
org.omg.CORBA.ORB orb = ORB.init(null, props);

// Resolve the RootPOA.
org.omg.CORBA.Object rootPoaRef =
   orb.resolve initial references(“RootPOA”);

org.omg.PortableServer.POA rootPoa =
```

```
   org.omg.PortableServer.POAHelper.narrow(rootPoaRef);
```

6.3.4.6 POAManager

In minimum CORBA, the POAManager has been strongly reduced and it is only used to activate the POAs it is associated with. If the POA has not been activated, no requests will be processed for the servants activated under it.

It is recommended to use the same POAManager instance for the POAs so that no POA will accidentally be in a non-activated state. A POAManager instance can be retrieved from the RootPOA when creating a new POA.

```
// Get the POAManager from the RootPOA.
org.omg.PortableServer.POAManager poam = rootPoa.the POAManager();
```

6.3.4.7 Code examples for creating POAs

In the following example, a persistent POA is created:

```
// Initialise the ORB.
Hashtable props = new Hashtable();
props.put("com.nokia.m2m.orb.UseM2MGateway", "no");
org.omg.CORBA.ORB orb = ORB.init(null, props);

// Get the RootPOA.
POA rootPoa =
   POAHelper.narrow(orb.resolve initial references("RootPOA"));

// Create policies for a persistent POA.
org.omg.CORBA.Policy[] persistentPolicies =
   new org.omg.CORBA.Policy[] {
      rootPoa.create id assignment policy(
         AssignmentPolicyValue.USER ID),
      rootPoa.create id uniqueness policy(
         IdUniquenessPolicyValue.UNIQUE_ID),
      rootPoa.create_lifespan_policy(
         LifespanPolicyValue.PERSISTENT)};

// Create the persistent POA. Retrieve the POAManager from the
// RootPOA and use the persistent policies.
POA persistentPOA = rootPoa.create_POA("PersistentPOA",
   rootPoa.the POAManager(), persistentPolicies);
```

In this example, a transient POA is created:

```
// Initialise the ORB.
Hashtable props = new Hashtable();
props.put("com.nokia.m2m.orb.UseM2MGateway", "no");
org.omg.CORBA.ORB orb = ORB.init(null, props);

// Get the RootPOA.
POA rootPoa =
   POAHelper.narrow(orb.resolve initial references("RootPOA"));

// Create policies for a transient POA.
```

```
org.omg.CORBA.Policy[] transientPolicies =
   new org.omg.CORBA.Policy[] {
      rootPoa.create id assignment policy(
         AssignmentPolicyValue.SYSTEM ID),
      rootPoa.create id uniqueness policy(
         IdUniquenessPolicyValue.UNIQUE_ID),
      rootPoa.create_lifespan_policy(
         LifespanPolicyValue.TRANSIENT)};

POA transientPOA = rootPoa.create POA("TransientPOA",
   rootPoa.the_POAManager(), transientPolicies);
```

6.3.4.8 Object key

The object key consists of the ORB's implementation name, POA path, and object ID. The RootPOA is not included in the object key.

For example, when :

- the POA has the name "ServerPOA" and it is created under the RootPOA,
- a servant is activated under the ServerPOA with an object ID "ExampleServant",
- and the ORB does not have an implementation name,

the POA path is 'RootPOA/ServerPOA', and the object key is 'ServerPOA/ExampleServant'.

If the ORB has, for example, 'J2MEORB' as the implementation name, the object key is 'J2MEORB/ServerPOA/ExampleServant'.

6.3.4.9 Object references

When an ORB publishes an IOR, it always contains at least the IIOP profile (IP address, port, and object key). The IOR can also contain other profiles. The J2ME ORB needs a negotiated IP address for the network link when it is publishing object references. To meet this requirement the network link must be open when the J2ME ORB creates the object reference. This generates a limitation in which object reference publishing is only possible in the invocation context. The invocation context refers to a situation in which a servant, located in an IMlet, is processing a server-originated request.

The reference can then be bound, for example, to the CORBA Naming Service. The CORBA client, located in the customer server application, can then retrieve the reference from the Naming Service if it knows the binding name.

```
// An object reference.
org.omg.CORBA.Object ref = null;

// ... Create the reference ...
```

```
// Resolve the CORBA Naming Service. The IP address has to
// be known.
org.omg.CORBA.Object nsRef =
orb.string to object("corbaloc::10.35.1.207:900/NameService");
NamingContextExt ns = NamingContextExtHelper.narrow(nsRef);

// Publish the reference with a name "ExampleService".
// This name can be used to resolve an IOR to the servant.
ns.rebind(ns.to name("ExampleService"), ref);
```

This code example connects to the CORBA Naming Server located in 10.35.1.206:900, and binds an object reference into the root context by using the name ‘ExampleService’. Now other CORBA applications can resolve the reference using that name as key.

6.3.4.10 Servant initialisation

The servant implementation is an implementation of the xxxPOA class formed by the IDL compiler from the interface defined in the IDL.

The same servant implementation is used here as in Chapter 6.2.2:

```
package hello.impl;

import hello.GreeterPOA;

public class GreeterImpl extends GreeterPOA
{
   // This method is invoked by the remote call.
   public String sayHello(String name)
   {
      return “Hello world and “ + name + “!”;
   }
}
```

An example for a transient POA (RootPOA):

```
// Initialise the ORB
Hashtable props = new Hashtable();
props.put("com.nokia.m2m.orb.UseM2MGateway", "no");
org.omg.CORBA.ORB orb = ORB.init(null, props);

// Get the RootPOA.
POA rootPoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));

// Create a new servant implementation.
GreeterImpl impl = new GreeterImpl();

// Activate the servant.
byte[] objectId = rootPoa.activate_object(impl);

// Activate the POAManager.
rootPoa.the_POAManager().activate();
```

The servant is now ready to process invocations from the client.

An example for a persistent POA:

```
// Initialise the ORB
Hashtable props = new Hashtable();
props.put("com.nokia.m2m.orb.UseM2MGateway", "no");
org.omg.CORBA.ORB orb = ORB.init(null, props);

// Get the RootPOA.
POA rootPoa = POAHelper.narrow(orb.resolve initial references("RootPOA"));


// Create policies for a persistent POA.
org.omg.CORBA.Policy[] persistentPolicies =
   new org.omg.CORBA.Policy[] {
      rootPoa.create id assignment policy(
         AssignmentPolicyValue.USER ID),
      rootPoa.create id uniqueness policy(
         IdUniquenessPolicyValue.UNIQUE ID),
      rootPoa.create_lifespan_policy(
         LifespanPolicyValue.PERSISTENT)};

// Create the persistent POA. Retrieve the POAManager from the
// RootPOA and use the persistent policies.
POA serverPOA = rootPoa.create_POA("ServerPOA",
   rootPoa.the_POAManager(), persistentPolicies);

// A new servant implementation.
GreeterImpl impl = new GreeterImpl();

// Activate the servant.
serverPoa.activate_object_with_id("HelloServer".getBytes(), impl);

//Activate the POAManager.
rootPoa.the_POAManager().activate();
```

6.3.4.11 Request from the server program

This issue is handled in the *Nokia M2M Platform Server Application Programming Guide.*

6.3.5 Callback

The client can pass a CORBA interface as a parameter in a remote method invocation, and the server can call a method on that remote object.

An example IDL where the server is implemented on the other side and the client on the other:

```
module callbackexample
{
   interface Client
   {
      void calledByServer(in string name);
   };
```

```
   interface Server
   {
      void takeReference(in Client callback);
   };
};
```

A server-side implementation of the generated *ServerPOA* class:

```
package callbackexample.impl;

import callbackexample.Client;
import callbackexample.ServerPOA;

public class ServerImpl extends ServerPOA
{
   public void takeReference(Client callback)
   {
      callback.calledByServer("Server calling back");
   }
}
```

A client-side servant implementation:

```
package callbackexample.impl;

import callbackexample.ClientPOA;

public class ClientImpl extends ClientPOA
{
   public void calledByServer(String name)
   {
      // Add your implementation here.
   }
}
```

Client doing the invocation:

```
// Initialise the ORB
Hashtable props = new Hashtable();
props.put("com.nokia.m2m.orb.UseM2MGateway", "no");
org.omg.CORBA.ORB orb = ORB.init(null, props);

callbackexample.impl.ClientImpl impl = new ClientImpl();
callbackexample.Client clientRef = null;
callbackexample.Server server = null;

// The steps of activating the servant ClientImpl, getting a
// reference to it from the POA, and narrowing it are omitted.

// Do the remote method invocation. This results in the
// server calling back on the ClientImpl through the CORBA interface.
server.takeReference(clientRef);
```

6.3.6 Advanced features

This chapter describes different ways of creating references with the J2ME ORB in addition to the corbaloc object URL that has been used so far in this document. For method details, please refer to Java documentation on the J2ME ORB.

6.3.6.1 Create reference

First, the proprietary `ORB.createReference()` can be used to create object references without using the corbaloc object URL:

```
// Initialise the orb and cast it to com.nokia.m2m.orb.org.omg.CORBA.ORB
orb = org.omg.CORBA.ORB.init(null, null);
com.nokia.m2m.orb.ORB norb = (com.nokia.m2m.orb.ORB) orb;

// An IOR with one IIOP profile is returned.
org.omg.CORBA.Object refOne =
   norb.createReference(
      "10.35.1.207",
      9999,
      "NameService".getBytes(),
      "IDL:omg.org/CosNaming/NamingContextExt:1.0",
      "1.2");
```

An example of the `ORB.createReference()` using TaggedComponents:

```
// TaggedComponents can also be added to an IOR.
// Here an example is given on how to publish
// codesets in an IOR.
org.omg.IOP.TaggedComponent comp =
com.nokia.m2m.portable.GIOPOutputStream.createCodeSets(
      GIOPOutputStream.CODESET ISO8859 1,
      GIOPOutputStream.CODESET UTF16);

org.omg.CORBA.Object refTwo =
   norb.createReference(
      "10.35.1.207",
      9999,
      "NameService".getBytes(),
      "IDL:omg.org/CosNaming/NamingContextExt:1.0",
      "1.2",
      new org.omg.IOP.TaggedComponent[]{comp});
```

6.3.6.2 Wchar and wstring data types

Wstrings and wchars (wide character sets) are needed when the characters used in invocations do not belong to the ASCII characters. The wide character data types only work with GIOP versions 1.1 and 1.2.

ServiceContext that includes the CodesetComponents is transmitted in GIOP request and reply messages and it defines the used codeset. Because codesets can be transmitted with each GIOP request and reply, they increase the amount of data transmitted.

The J2ME ORB supports the following codesets for the wchar data type: UTF-8, UTF-16, and ISO8859-1. For the char data type, the only supported codeset is ISO8859-1.

Codesets published by the J2ME ORB:

- char: ISO8859-1 default, UTF-8 conversion codeset
- wchar: UTF-16 default, no conversion codesets

Codesets can be added to the CORBA object reference at reference creation phase. The methods for creating references in the J2ME ORB enable the adding of CodesetComponents.

```
// Create the wanted codesets.
org.omg.IOP.TaggedComponent comp =
com.nokia.m2m.portable.GIOPOutputStream.createCodeSets(
      GIOPOutputStream.CODESET ISO8859 1,
      GIOPOutputStream.CODESET UTF16);

// Create a reference that includes a CodesetComponent.
org.omg.CORBA.Object refSix =
   norb.createNamedReference(
      new NameComponent[] { new NameComponent("HelloServer", "")},
      "IDL:hello/Greeter:1.0",
      "10.35.1.207",
      9251,
      new org.omg.IOP.TaggedComponent[]{comp});
```

An example IDL that uses the wstring and wchar data types:

```
module example
{
   interface WideService
   {
      wchar wcharExample(in wchar wideChar);
      wstring wstringExample(in wstring wideString);
   };
};
```

	**Note:** Wide character set implementations in different ORBs that use GIOP version 1.1 are not always compatible. Because wstring and wchar data types add more complexity in the application, you should avoid using them if they are not needed by the application. When using wide character types, GIOP version 1.2 is

	highly recommended. The J2ME ORB uses GIOP version 1.2 by default.

6.3.6.3 Any data type

The J2ME ORB supports the CORBA `Any` data type. As the name implies, it can hold any CORBA data type.

An example IDL:

```
module anyexample
{
   interface AnyHandler
   {
      any testAny(inout any exampleAny);
   };
};
```

The IDL compiler generates an AnyHandler CORBA interface. The inout Any requires that the AnyHolder is used. The server then handles the inout data.

Client-invocation:

```
// ORB initialisation and reference creation are omitted.

anyexample.AnyHandler anyhandler = anyexample.AnyHandlerHelper.narrow(ref);

// Create Any. This can be done via the singleton -ORB.
org.omg.CORBA.Any str = org.omg.CORBA.ORB.init().create any();

// Insert a string to the Any and set the Any into the holder.
str.insert string("Hello World!");
org.omg.CORBA.AnyHolder holder = new org.omg.CORBA.AnyHolder(str);

org.omg.CORBA.Any response = anyhandler.testAny(holder);
org.omg.CORBA.Any holderAny = holder.value;

// Extract the values.
String responseValue = response.extract_string();
String holderValue = holderAny.extract_string();
```

For more examples on using the Any data type, please refer to Chapter 7.

7. NOKIA 12 MODULE ORB SERVICES

This chapter describes the Module ORB services available via the CORBA interface. The same services can also be used from an AM and a customer server application over a wireless link.

Using the Module ORB services from an IMlet is very straightforward. Stubs and skeletons from the Nokia 12 module interface definitions have already been compiled from the interface definition, and they have been included into the system classes (**classes.zip**). Thus, you do not have to compile them or include them in an IMlet Suite JAR file.

Describing the Module ORB services focuses on how they can be used from IMlets. Instead of describing all available methods, simple example code snippets are included. For more information on Module ORB service, please refer to the *Nokia 12 GSM Module Interface Definition Reference Guide* and *Nokia 12 GSM Module Properties Reference Guide*.

Note: It is highly recommend that Module ORB is set to IIOP mode to allow it to communicate with standard IIOP ORBs. You can change Module ORB mode with the *ModuleORBMode* configuration parameter. See chapter Appendix C: An example of configuring the Nokia 12 GSM module from IMlet for the configuration example code.

7.1 ACCESSING MODULE ORB SERVICES

The following steps are needed to access the Module ORB services. The example given here describes the use of the IOModule, but the same pattern can also be applied to other services by changing the object key and stub classes that are used.

1. Initialise the ORB. This initialisation sets the J2ME ORB into IIOP mode.

```
// Initialise the ORB
Hashtable props = new Hashtable();
props.put("com.nokia.m2m.orb.UseM2MGateway", "no");
org.omg.CORBA.ORB orb = ORB.init(null, props);
```

2. Create an object reference to the desired service. The simplest way to do this is to use `corbaloc` URL as described below. The object key and address of the servant are needed to create the reference. The object keys for different services are listed in the following chapters. The address is always 127.0.0.1:19740.

```
String url="corbaloc::127.0.0.1:19740/ORB/OA/IDL:IOModule/IOControl:1.0";
```

```
org.omg.CORBA.Object ref = orb.string_to_object(url);
```

3. Narrow the returned `org.omg.CORBA.Object` to a specific type. Use helper classes to narrow the reference.

```
com.nokia.m2m.orb.idl.iomodule.IOControl iocontrol =
      com.nokia.m2m.orb.idl.iomodule.IOControlHelper.narrow(ref);
```

4. Call the methods. Catch possible CORBA application exceptions as required by the used interface definition. It is a good practice to also catch CORBA SystemExceptions even though the communication in this case is local.

```
   try {

      // Set output pin 7 on
      iocontrol.setOutputPinValue((short) 7, 1);

   } catch (PinValueOutOfRange valueOutOfRange) {
      log(valueOutOfRange.toString());
   } catch (PinNumberOutOfRange numberOutOfRange) {
      log(numberOutOfRange.toString() );
   } catch (SystemException se) {

      log(se.toString() + " minor=" + se.minor);
   }
```

If executing the above code throws a `CORBA OBJECT_NOT_EXISTS` exception, check that the object key value is correct.

7.2 WIRELESS DEVICE

This service is used for retrieving basic device information and for obtaining, setting, and observing device properties.

Stubs: `com.nokia.m2m.orb.idl.wirelessdevice`

`com.nokia.m2m.orb.idl.wirelessdevice.DevicePackage`

Interface: `com.nokia.m2m.orb.idl.wirelessdevice.Device`

Helper: `com.nokia.m2m.orb.idl.wirelessdevice.DeviceHelper`

Object key: `Device`

7.2.1 An example of using the getDeviceInfo

The following example shows how to resolve and print out device information. `DeviceInfo` contains information about the hardware and software versions of the module. This example also shows how to handle CORBA 'out' type arguments in method calls.

For each 'out' type argument there is a corresponding holder class, named as `<TypeName>Holder`. The CORBA classes in the Nokia 12 module include holder classes for all basic data types (such as `LongHolder` for long and `StringHolder` for String). The IDL compiler generates the holder classes for the user-defined data types.

```
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(null, null);

//
// Resolve the Device object.
//
String url = "corbaloc::127.0.0.1:19740/Device";
org.omg.CORBA.Object ref = orb.string_to_object(url);
Device device = DeviceHelper.narrow(ref);

/*
   Related IDL goes as follows:

   typedef sequence<string>    MetadataList;

   struct Version {
      octet    major;
      octet    minor;
      short    build;
      long     date;
   };

   void getDeviceInfo(
      out string  hwManufacturer,
      out string  hwProductName,
      out Version hwVersion,
      out string  swManufacturer,
      out string  swProductName,
      out Version swVersion,
      out MetadataList metadata );

*/


//
// Create holders for 'out' arguments.
//
StringHolder hwManufacturer = new StringHolder();
StringHolder hwProductName = new StringHolder();
VersionHolder hwVersion = new VersionHolder();

StringHolder swManufacturer = new StringHolder();
StringHolder swProductName = new StringHolder();
VersionHolder swVersion = new VersionHolder();

MetadataListHolder metadata = new MetadataListHolder();

//
// Invoke the method.
//
device.getDeviceInfo(
   hwManufacturer,
```

```
      hwProductName,
      hwVersion,
      swManufacturer,
      swProductName,
      swVersion,
      metadata);

   //
   // Print the results.
   //
   String deviceInfo = "hwManufacturer=" + hwManufacturer.value + "\n";
   deviceInfo += "hwProductName=" + hwProductName.value + "\n";
   deviceInfo += "hwVersion="
            + hwVersion.value.major
            + "."
            + hwVersion.value.minor
            + " build "
            + hwVersion.value.build
            + "\n";

   deviceInfo += "swManufacturer=" + swManufacturer.value + "\n";
   deviceInfo += "swProductName=" + swProductName.value + "\n";
   deviceInfo += "swVersion="
            + swVersion.value.major
            + "."
            + swVersion.value.minor
            + " build "
            + swVersion.value.build
            + "\n";

   log(deviceInfo);
```

7.2.2 An example of using the HTTP parameters

One of the most important features of the Wireless Device service is the possibility to change the configuration of the Nokia 12 module. When configuration is changed using the `setParam()` method, the changes are stored into non-volatile memory and they remain valid even when the Nokia 12 module is restarted. The `getParam()` method is available for reading the configuration values of the Nokia 12 module.

The example below shows how to set HTTP parameters to the Nokia 12 module. The example shows how to construct the more complex data structures that are used when setting the HTTP parameters. Similar data structures are needed when M2M wireless bearers are set. This code snippet is from the `com.nokia.m2m.sdk.examples.http` package. The `com.nokia.m2m.sdk.examples.http` package and the `HTTPParametersIMlet` are included in the Nokia 12 SDK.

```
public void startApp() throws MIDletStateChangeException {

      try {

         log("Starting HTTPParametersIMlet");

         // Initialise the ORB
         Hashtable props = new Hashtable();
         props.put("com.nokia.m2m.orb.UseM2MGateway", "no");
         org.omg.CORBA.ORB orb = ORB.init(null, props);
```

```
// Get the Device object.
String url = "corbaloc::127.0.0.1:19740/Device";
org.omg.CORBA.Object devRef = orb.string_to_object(url);
Device dev = DeviceHelper.narrow(devRef);

/*
 * struct HTTPProxyInformation {
 *   boolean           proxyEnabled;
 *   unsigned short    primaryProxyPort;
 *   unsigned short    secondaryProxyPort;
 *   string            primaryProxyAddress;
 *   string            secondaryProxyAddress;
 * };
 *
 */

// Do not use a proxy.
HTTPProxyInformation httpProxy =
   new HTTPProxyInformation(false, (short) 0, (short) 0, "", "");

/*
 * struct HTTPBearerInformationGPRS {
 *  HTTPAuthenticationType    authenticationType;
 *  HTTPContextOpenType       contextOpenType;
 *  HTTPLoginType             loginType;
 *  string                    GPRSAccessPointName;
 *  string                    GWIPAddress;
 *  string                    userName;
 *  string                    passWord;
 * };
 *
 */

HTTPBearerInformationGPRS gprsBearer =
   new HTTPBearerInformationGPRS(
      HTTPAuthenticationType.CHAP,
      HTTPContextOpenType.ON DEMAND,
      HTTPLoginType.AUTOMATIC,
      "internet",
      "",
      "",
      "");

/*
 * union HTTPBearerInformationUnion switch (HTTPBearerType) {
 *  case HTTP CSD:    HTTPBearerInformationCSD     bearerInfoCSD;
 *  case HTTP_GPRS:   HTTPBearerInformationGPRS    bearerInfoGPRS;
 * };
 *
 */

HTTPBearerInformationUnion bearerInformation =
   new HTTPBearerInformationUnion();
bearerInformation.bearerInfoGPRS(gprsBearer);

//
// When all components are ready, they
// are set into the HTTPInformation structure.
//

/*
 * struct HTTPInformation {
 *   HTTPSessionType               HTTPActiveSessionType;
 *   HTTPBearerType                HTTPActiveBearerType;
 *   HTTPSessionSecurity           HTTPSessionSecurity;
 *   HTTPProxyInformation          HTTPProxyInformation;
 *   HTTPBearerInformationUnion    HTTPBearerInformationUnion;
 * };
 *
```

```
         */

        HTTPInformation info =
           new HTTPInformation(
              HTTPSessionType.CO,
              HTTPBearerType.HTTP_GPRS,
              HTTPSessionSecurity.UNSECURE,
              httpProxy,
              bearerInformation);

        // Insert the constructed structure into CORBA Any.
        Any any = orb.create_any();
        HTTPInformationHelper.insert(any, info);

        dev.setParam("HTTPInformation", any);

        log("HTTPParameters set");
      } catch (Exception e) {
        log(e.toString());
      }
   }
```

After running this IMlet you can check the configuration change by using the Nokia 12 Configurator.

7.3 EMBEDDED TERMINAL

This service is used for controlling the basic GSM functionalities (such as PIN codes, SMS messages, Phonebook, and GSM connections).

Stubs: `com.nokia.m2m.orb.idl.terminal`

`com.nokia.m2m.orb.idl.terminal.ETPackage`

Interface: `com.nokia.m2m.orb.idl.terminal.ET`

Helper: `com.nokia.m2m.orb.idl.terminal.ETHelper`

Object key: `ORB/OA/IDL:ET:1.0`

Tip: The International Mobile Equipment Identity (IMEI) property in the Embedded Terminal interface can be used to identify the Nokia 12 module. If your IMlet needs to resolve its own identify, it can read the IMEI by calling the `ET.IMEI()` method. All GSM devices have a unique IMEI code set by the manufacturer.

7.3.1 An example of reading short messages from a SIM card

The example code presented in this chapter reads short messages from a SIM card. After a message has been read, the IMlet writes message data to the serial port and removes the message from the SIM card. This approach is based on calling the `et.readShortMessage()` frequently and catching raised exceptions if there are no short messages to read.

Another, and more elegant way to receive short messages is to register an observer for the SMS-RECEIVED-STORED event. The event observer approach consumes less CPU resources because a read attempt is made only after the event observer has indicated that it has received a short message. The use of event observers is described in Chapter 7.7

```
//Initialise the ORB
 Hashtable props = new Hashtable();
 props.put("com.nokia.m2m.orb.UseM2MGateway", "no");
 org.omg.CORBA.ORB orb = ORB.init(null, props);

 // Resolve the Device object.
 String url = "corbaloc::127.0.0.1:19740/ORB/OA/IDL:ET:1.0";
 org.omg.CORBA.Object ref = orb.string_to_object(url);

 ET et = ETHelper.narrow(ref);

 // Prepare the arguments.
 StringHolder msg = new StringHolder();
 StringHolder sender = new StringHolder();

 while (true) {
    try {

       // Throws an exception if there is no message.
       et.readShortMessage(MessageStorage.SIM, 0, msg, sender);

       // Message received.
       log("Received message: " + msg.value + " from: "+ sender.value);

       // Remove message.
       et.removeShortMessage(MessageStorage.SIM,0);

    } catch (Exception ignored) {
       // wait one second before reading again.
       try {
          Thread.sleep(1000);
       } catch (InterruptedException interrupted) {}
    }
 }
```

7.4 IO MODULE

This service is used for controlling and observing the I/O pins of the Nokia 12 module.

	**Note:** Do not confuse the IO Module service with the IOControl Java API. These are optional interfaces that affect the same I/O pins.

Stubs: `com.nokia.m2m.orb.idl.iomodule`

Interface: `com.nokia.m2m.orb.idl.iomodule.IOControl`

Helper: `com.nokia.m2m.orb.idl.iomodule.IOControlHelper`

Object key: `ORB/OA/IDL:IOModule/IOControl:1.0`

An example of using the IO Module is provided in Chapter 7.1.

7.5 GPS

This service can be used to read the Global Positioning System (GPS) data from an external GPS device that is physically attached to the Nokia 12 module. Before using this API, the GPS device must be attached to serial port 1 or 3. Also, the selected serial port must be configured to use the GPS mode.

	**Note:** Remember to set the other GPS parameters, such as serial port, speed, and National Marine Electronics Association (NMEA) parameters, according to the instructions of your GPS device.

Stubs: `com.nokia.m2m.orb.idl.gps`

Interface: `com.nokia.m2m.orb.idl.gps.GpsModule`

Helper: `com.nokia.m2m.orb.idl.gps.GpsModuleHelper`

Object key: `ORB/OA/IDL:gps/GpsModule:1.0`

7.5.1 An example of using the GPS service

A GPS example IMlet `com.nokia.m2m.sdk.examples.gps.GpsIMlet` is included in the Nokia 12 SDK.

The example IMlet reads GPS data every 30 seconds using the GPS CORBA API. For brevity, only the `startApp()` method is displayed here.

	**Note:** Since this IMlet uses serial port 3 to print log information to your workstation, configure the Nokia 12 module so that serial port 1 is set for the GPS device.

```
public void startApp() throws MIDletStateChangeException {
      try {
         log("GpsIMlet starting");

         // Initialise the ORB
         Hashtable props = new Hashtable();
         props.put("com.nokia.m2m.orb.UseM2MGateway", "no");
         org.omg.CORBA.ORB orb = ORB.init(null, props);

         // Get refrence to the GPS module.
         String url =
            "corbaloc::127.0.0.1:19740/ORB/OA/IDL:gps/GpsModule:1.0";
         org.omg.CORBA.Object ref = orb.string to object(url);
         GpsModule gpsModule = GpsModuleHelper.narrow(ref);

         String gpsInformation;
         String latitude;
         String longitude;
         while (true) {
            try {

               GpsData data = gpsModule.getGpsData();

               // Handle GPS data.
               gpsInformation =
                  "Visible satellites " + data.visibleSatellites + "\n";
               gpsInformation += "Altitude" + data.alt + "\n";

               latitude = data.position.lat.degrees + " degrees ";
               latitude += data.position.lat.minutes + " minutes ";
               latitude += data.position.lat.seconds + " seconds ";
               latitude += data.position.lat.milliseconds
                  + " milliseconds ";
               latitude += data.position.lat.type == NORTH.value
                  ? "North"
                  : "South";
               latitude += "\n";

               longitude = data.position.lon.degrees + " degrees ";
               longitude += data.position.lon.minutes + " minutes ";
               longitude += data.position.lon.seconds + " seconds ";
               longitude += data.position.lon.milliseconds
                  + " milliseconds ";
               longitude += data.position.lon.type == EAST.value
                  ? "East"
                  : "West";
               longitude += "\n";

               SerialPortLogger.getInstance().write(
                  gpsInformation + latitude + longitude);

            } catch (NoGpsDataAvailableException noGpsData) {
               // This exception is raised if the GPS device is
               // disconnected.
               log(noGpsData.toString());
            }

            Thread.sleep(30000);

         }
      } catch (Exception e) {
         log("Error: " + e.toString());

      }
   }
```

7.6 IMLET SUITE MANAGER

The IMlet Suite Manager service is used for controlling the life cycle of IMlet Suites inside the Nokia 12 module. The service can be used, for example, to load IMlets to the Nokia 12 module, to start and stop them, or to remove them from the Nokia 12 module.

For the IMlet itself the most useful feature of the IMlet Suite Manager service is the ability to start a new IMlet. When a new IMlet is started, the calling IMlet will be stopped automatically.

The IMlet Suite Manager service can also be used to stop an IMlet that is running.

Note: When IMlet Suite Manager service is used to stop an IMlet, the start-up record of the IMlet in question is cleared and the IMlet will not start after the next reboot.

Stubs: `com.nokia.m2m.orb.idl.imletmanager`

Interface: `com.nokia.m2m.orb.idl.imletmanager.SuiteManager`

Helper: `com.nokia.m2m.orb.idl.imletmanager.SuiteManagerHelper`

Object key: `ORB/OA/IDL:SuiteManager:1.0`

7.6.1 An example of using the IMlet Suite Manager

This example code resolves the `SuiteManager` object and then stops the running IMlet. An IMlet that is running can also be stopped by first calling the `getActiveIMlets()` method from the `SuiteManager`, and then stopping the returned IMlet.

```
public void startApp() throws MIDletStateChangeException {
      try {
         log("StopDemo starting");

         org.omg.CORBA.ORB orb = ORB.init(null, null);

         // Get a reference to the SuiteManager.
         String url =
            "corbaloc::127.0.0.1:19740/ORB/OA/IDL:SuiteManager:1.0";

         org.omg.CORBA.Object ref = orb.string to object(url);
         SuiteManager suiteManager = SuiteManagerHelper.narrow(ref);

         // Resolve the IMlet Suite and IMlet.
         Suite suite=suiteManager.getSuite("Examples");
         IMlet me=suite.getIMlet("StopDemo");
```

```
            log("Waiting for 30 seconds...");
            Thread.sleep(30000);

            log("Stopping...");
            me.stop();

            // This row will never be reached because the IMlet
            // has already been stopped.
            log("Still running!");

        } catch (Exception e) {
            log("Error: " + e.toString());

        }
    }
```

7.7 OBSERVERS AND CALLBACKS

Some services in the Module ORB support a mechanism that makes it possible to register CORBA objects as event observers. Using observers removes the need to do frequent polling for certain methods.

The Wireless Device service allows an observer to listen to events and certain parameters. The IO Module service makes it possible to monitor input pins, and the IMlet Suite Manager service can be used to monitor the running status of an IMlet.

IMlets themselves can also utilise observers set to the IO Module or Wireless Device. Events that can be observed are, for example, call status, network signal quality, and incoming short messages. The available parameters, events, and their structures are explained in the *Nokia GSM Connectivity Terminal And Nokia 12 GSM Module Properties Reference Guide*

To listen to events you must first implement a servant and activate the servant into the J2ME ORB. After the servant (and the POA managing the servant) is ready and active, the servant can be registered into a Module ORB service as an observer.

When an event is fired, the service in the Module ORB creates a callback to the J2ME ORB, and the J2ME ORB routes the callback to the servant. Every event handler method is defined so that the event name is placed as an argument to the method call. Some events include data payload as well, whereas some events rely solely on the information carried in the event name. If an event has data payload, it is provided in the CORBA `Any` structure that is specified in the interface where the event is defined. The observer implementation knows the structure of `Any` by checking the event name. Based on this name, use the correct `Helper` class to extract data from `Any`.

The same pattern can be used to listen to events from all different services provided by the Module ORB. Every service has its own observer interfaces

that must be implemented when observing those services, but the overall framework is always the same.

Restarting the Nokia 12 module does not remove observers. Thus it is important to use persistent servants as listener objects. When a persistent servant is used, an IMlet installs the same IOR as the listener object every time it starts.

If transient servants are used, an IMlet must ensure that it removes the existing observers before installing new ones. The number of observers in the Nokia 12 module is limited. The maximum number of supported observers is 10. The first four observers are stored into the persistent memory and they stay valid when the Nokia 12 module restarts. Other observers are lost when the Nokia 12 module restarts. One observer can be set to monitor multiple events and thus these limits can be avoided by properly planning the use of observers.

Tip: You can start listening to all events by giving an empty string "" as the event name in the `setEventObserver()` method. However, the same pattern cannot be used to listen to all parameter-related observers.

Caution: The `Device.removeAllObservers()` method removes all observers from the Nokia 12 module. If an IMlet calls this method, the observers set from AMs and customer server applications are also removed. Observers are also removed from the IMlet if an AM or a customers server application calls the `removeAllObservers()` method.

The use of observers generates traffic between the Module ORB and J2ME ORB. Thus it is recommended that the application be set to only listen to the events that it needs.

7.7.1 An example of using an observer with the SignalQuality parameter

The following example code has two classes. One is the `SignalQualityIMlet` class, which starts the J2ME ORB, activates the observer servant, and registers the servant to the J2ME ORB. The other is the `SignalQualityObserverImpl` class, which is registered as an observer and handles the incoming events. When these classes have been initialised, the IMlet keeps running and the observer listens to incoming parameter change events. In this example only the `SignalQuality` parameter is monitored.

Note: `SignalQuality` and other parameters can also be asked from the Nokia 12 module using the `Device.getParam()` method.

The observer servant is described in the first code listing. This servant is inherited from the

`com.nokia.m2m.orb.idl.wirelessdeviceParamObserverPOA` class and, as the interface definition specifies, it must implement the `paramChanged()` method. The J2ME ORB calls this method when it receives a callback request from the Module ORB. You can implement the servant code and process the data sent in the event as needed.

An instance of the `SignalQualityObserverImpl` class is created in the `SignalQualityIMlet` listing below, and it is registered as a parameter observer into the Wireless Device service.

```
package com.nokia.m2m.sdk.examples.device;

import org.omg.CORBA.Any;
import
com.nokia.m2m.orb.idl.properties.ParametersPackage.SignalQualityHelper;
import com.nokia.m2m.orb.idl.wirelessdevice.ParamObserverPOA;
import com.nokia.m2m.sdk.examples.logger.SerialPortLogger;


/**
 * This class implements the ParamObserver interface via inheriting the
 * ParamObserverPOA.
 *
 *
 */
public class SignalQualityObserverImpl extends ParamObserverPOA {

   public void paramChanged(String paramName, Any oldValue, Any newValue) {

      //
      // Check that the parameter is correct before
      // trying to extract values from Any.
      //
      if(paramName.equals("SignalQuality")){

         //
         // Extract values from Any.
         //
         short oldVal= SignalQualityHelper.extract(oldValue);
         short newVal=SignalQualityHelper.extract(newValue);

         log("SignalQuality, oldValue=" + oldVal + " newValue=" + newVal);

      }else {
         log("Unknown paramChanged event: " + paramName);
      }
   }

   private void log(String msg){
      SerialPortLogger.getInstance().write(msg);
   }

}
```

The following code registers the created `SignalQualityObserverImpl` as the listener for the `SignalQuality` parameter.

```
package com.nokia.m2m.sdk.examples.device;
```

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;

import org.omg.CORBA.Object;
import org.omg.CORBA.Policy;

import org.omg.PortableServer.IdAssignmentPolicyValue;
import org.omg.PortableServer.IdUniquenessPolicyValue;
import org.omg.PortableServer.LifespanPolicyValue;
import org.omg.PortableServer.POA;
import org.omg.PortableServer.POAHelper;

import com.nokia.m2m.orb.idl.wirelessdevice.Device;
import com.nokia.m2m.orb.idl.wirelessdevice.DeviceHelper;
import com.nokia.m2m.orb.idl.wirelessdevice.ParamObserverHelper;

import com.nokia.m2m.sdk.examples.logger.SerialPortLogger;

public class SignalQualityIMlet extends MIDlet {

   /**
    * This IMlet demonstrates the use of the Device CORBA interface.
    * It sets the parameter observer to observe SignalQuality changes.
    */
   public void startApp() throws MIDletStateChangeException {
      try {
         log("SignalQualityIMlet starting");

         // Initialise the ORB.
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(null, null);

         //
         // Get the RootPOA.
         //
         org.omg.PortableServer.POA rootPoa;
         rootPoa =
            POAHelper.narrow(orb.resolve initial references("RootPOA"));

         //
         // Create a new POA. Use persistent policies
         // so that servants activated under
         // this POA will always get the same IOR. This ensures
         // that when the IMlet is restarted
         // after Nokia 12 module reboot, the new reqistration will
         // overwrite the existing reqistration.
         //
         Policy[] policies =
            new Policy[] {
               rootPoa.create id assignment policy(
                  IdAssignmentPolicyValue.USER ID),
               rootPoa.create id uniqueness policy(
                  IdUniquenessPolicyValue.UNIQUE ID),
               rootPoa.create_lifespan_policy(
                  LifespanPolicyValue.PERSISTENT)};

         POA observerPoa =
            rootPoa.create POA(
               "ObserverPOA",
               rootPoa.the_POAManager(),
               policies);

         //
         // Initialize the servant object.
         //
         SignalQualityObserverImpl observerImpl =
            new SignalQualityObserverImpl();
         observerPoa.activate object with id(
            "SignalQualityObserver".getBytes(),
            observerImpl);
```

```
        //
        // Activate the POAManager.
        // Activation starts request processing in all POAs that
        // use this POAManager.
        //
        rootPoa.the_POAManager().activate();

        //
        // The observer is now ready to receive events.
        // Register the listener into the Module ORB.
        //

        // Resolve the Device object.
        String url = "corbaloc::127.0.0.1:19740/Device";
        org.omg.CORBA.Object ref = orb.string_to_object(url);

        Device device = DeviceHelper.narrow(ref);
        org.omg.CORBA.Object observerObj=
           observerPoa.servant_to_reference(observerImpl);

        log("Setting observer");

        // Start to listen to the SignalQuality parameter.
        device.setParamObserver(
           "SignalQuality",
           ParamObserverHelper.narrow(observerObj));

        log("Observer set. Waiting for events");

        orb.run();

      } catch (Exception e) {
        log("Error: " + e.toString());

      }
   }

   private void log(String msg) {
      SerialPortLogger.getInstance().write(msg);
   }

   public void pauseApp() {
   }

   public void destroyApp(boolean arg0) throws MIDletStateChangeException {
   }

}
```

When this code is executed, it prints the signal quality into the serial port. Frequency of events depends on the rate of signal quality changes. A typical frequency is a few events per minute.

8. EXAMPLE APPLICATION

This chapter describes an RMSLogger example application. The RMSLogger is a logging system that uses the RMS storage to store log information. It provides a CORBA interface for remotely reading and deleting log information, and a Java API for the IMlet to store log information.

Full example code is available in the `com.nokia.m2m.sdk.examples.logger.RMSLogger` class provided in the Nokia 12 SDK.

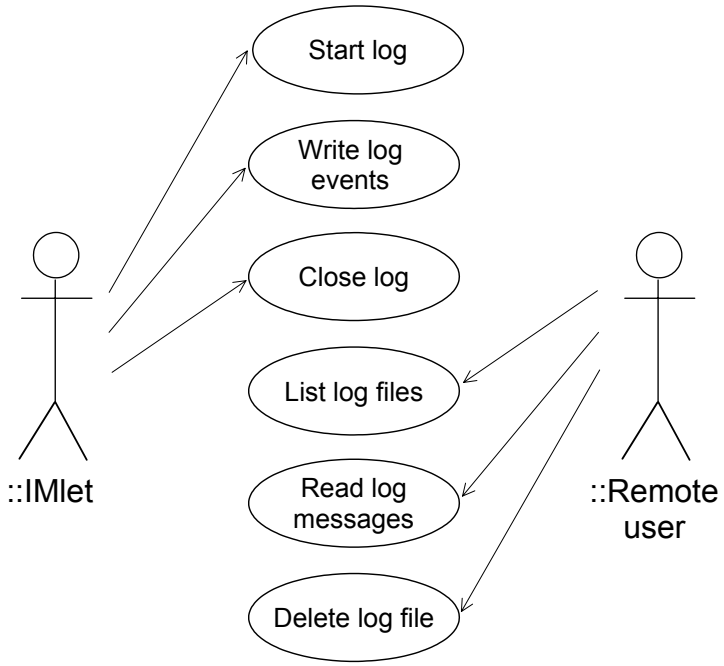

Figure 6. RMSLogger use cases

Two different interfaces are needed. One (Java API) is for the IMlet and it is used to print log messages into the log. The other (CORBA API) is for the remote application and it is used to read log messages from the log. The `RMSLogger` class implements both interfaces.

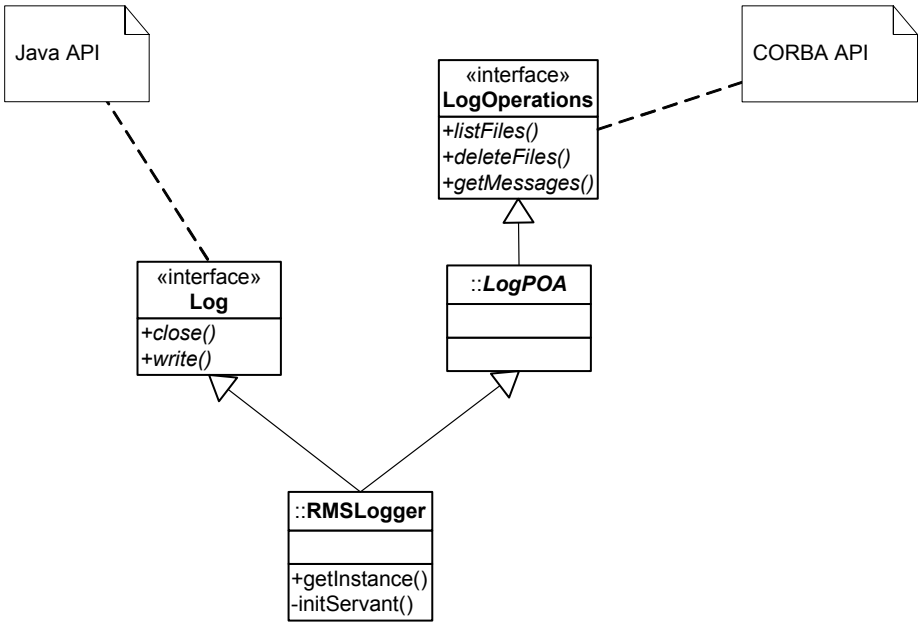

Figure 7. Class diagram

8.1 JAVA API

The Java API is described in the following code listing. The `SerialPortLogger` uses the same interface.

RMSLogger Java API:

```
package com.nokia.m2m.sdk.examples.logger;


public interface Log {

    /**
     * Writes a message to the log.
     * @param msg Message to be written
     */
   public void write(String msg);

   /**
    * Closes the log. Frees all resources used by the logger.
    *
    */
   public void close();

}
```

The `RMSLogger` class is a singleton and it is used by getting a reference using the `RMSLogger.getInstance()` method. The same class implements both Java and CORBA APIs.

RMSLogger singleton:

```
package com.nokia.m2m.sdk.examples.logger;
import javax.microedition.rms.RecordStore;

public class RMSLogger extends LogPOA implements Log {

   private static RMSLogger instance;

   private RMSLogger(ORB orb) {
      this.orb = orb;
      initServant();
   }

         ...


   /**
    * Returns an instance of the SerialLogger.
    * @return
    */
   public static synchronized Log getInstance(ORB orb) {
      if (instance == null) {
         instance = new RMSLogger(orb);
      }
      return instance;
   }


         ...

}
```

The RMSLogger is used from an IMlet as follows:

```
RMSLogger.getInstance(orb).write("This is a log message");
```

Note: Many examples in this document use either the SerialPortLogger or RMSLogger.

8.2 CORBA INTERFACE

The following operations are needed for remote use: listing log files, reading log messages, and deleting log files. The IDL presented here provides all these functionalities.

CORBA interface definition:

```
module logger{

   typedef sequence<string> StringList;
```

```
   struct LogMessage{
      long long   timestamp;
      string      data;
   };

   typedef sequence<LogMessage> MessageList;
   exception LogException{
       string reason;
   };

   interface Log{

      // List available log files
      StringList listFiles() raises (LogException);

      // Delete the log file
      void deleteFile(in string file) raises (LogException);

      // Return messages from the log file
      MessageList getMessages(in string fileName) raises (LogException);

   };
};
```

Compile the IDL using the idlj compiler from Sun J2SE SDK.

IDL compilation from the command line:

```
idlj -fall -pkgPrefix logger com.nokia.m2m.sdk.examples.logger.idl -td
compiled logger.idl
```

This command compiles the **logger.idl** file and places the resulted stub and skeleton files into the `com.nokia.m2m.sdk.examples.logger.idl` package. The next step is to modify the generated files to make them J2ME-compatible, and include them in IMlet sources. Modifications needed to helper and stub classes are explained in Chapter 6.1.2.2.

After the IDL is compiled, the `RMSLogger` class is defined to inherit the LogPOA. This requires implementing the three methods defined in the IDL to the `RMSLogger` class.

RMSLogger methods that are waiting for implementation:

```
   /**
    *
    * StringList listFiles() raises (LogException);
    */
   public String[] listFiles() throws LogException {

   }

   /**
    *
    * void deleteFile(in string file) raises (LogException);
```

```
     */
    public void deleteFile(String file) throws LogException {

    }

    /**
     *
     * MessageList getMessages(in string fileName) raises (LogException);
     */
    public LogMessage[] getMessages(String fileName) throws LogException {

    }
```

Servant initialisation and activation is needed before these methods can be called from remote applications. The `initServant()` method that is called from the constructor, handles the servant initialisation.

The `InitServant()` method creates a persistent POA for the RMSLogger servant and activates the servant into that POA. Because the POA is created using persistent policies, the resulted object key is known. The created object key is ‘logger/RMSLogger.

Servant initialisation:

```
private void initServant() {

      try {

         Object rpRef = orb.resolve initial references("RootPOA");
         POA rootPoa = POAHelper.narrow(rpRef);

         Policy[] policies =
            new Policy[] {
               rootPoa.create id assignment policy(
                  IdAssignmentPolicyValue.USER ID),
               rootPoa.create id uniqueness policy(
                  IdUniquenessPolicyValue.UNIQUE ID),
               rootPoa.create_lifespan_policy(
                  LifespanPolicyValue.PERSISTENT)};

         POA loggerPoa =
            rootPoa.create POA(
               "logger",
               rootPoa.the_POAManager(),
               policies);

         loggerPoa.activate object with id("RMSLogger".getBytes(), this);

         // The object key is now "logger/RMSLogger".

         // Make sure that the POAManager is active.
         rootPoa.the POAManager().activate();

      } catch (Exception ignored) {

      }

   }
```

The Logger CORBA interface is now ready to be called from remote applications.

APPENDIX A: BEARER PARAMETERS

This appendix guides you through the setting of wireless bearer parameters from an IMlet to the Nokia 12 module.

These parameters can be set either using the Nokia 12 Configurator or the Wireless Device CORBA API from an IMlet, AM, or remote server application.

The example IMlet described in this appendix has multiple methods; each responsible for setting one parameter structure using the `setParam()` method from the Wireless Device service. Due to backwards compatibility, these parameters are named as Wireless Application Protocol (WAP) parameters even though the Nokia 12 module does not use WAP.

When parameters are set using the CORBA API provided by the Wireless Device service, they are stored into persistent memory and stay valid even when the Nokia 12 module is restarted. The IMlet can read the parameters using the `getParam()` method.

Some figures from the Nokia 12 Configurator are included to demonstrate how the parameter values entered in the code are shown in the Configurator UI. Please note that all parameters that can be set from the Configurator UI can also be set from the IMlet.

The example IMlet described in this appendix is included in the Nokia 12 SDK.

The following BearerSettingsIMlet is located in the package com.nokia.m2m.sdk.examples.device:

```
public class BearerSettings extends MIDlet {

   // PPP username   and password
   final String pppUserName = "username";
   final String pppPassword = "secret";
   final String gprsAccesPointName = "ap.example.com";

   // Modem pool dial number. This is needed if the CSD bearer is used
   final String csdModemPoolNumer = "";

   // Gateway address is required only if wake-up service is used and
   // it can be left empty in most cases.
   //final String gwIPAddress = "10.35.1.195";
   final String gwIPAddress = "";
   final short gwPort = 0;

   // Connection timeout. This parameter is not needed for GPRS,
   // but it is recommended to set it for CSD
   final int timeoutGprs = 0;
   final int timeoutCsd = 120;

   // Home Location Agent port address. This value is not used in
   // IIOP mode.
   // final String HLAAddress = "10.35.1.201";
   final String HLAAddress = "";
   final short HLAPort = 0;
```

```
   // Any address can be used.
   final String localIpAddress="10.35.1.5";


   private ORB orb = null;
   private Device device = null;

   public void startApp() throws MIDletStateChangeException {

      try {

         log("Starting BearerSettings");

         // Initialize the ORB and get the Device object.
         orb = ORB.init(null, null);

         String url = "corbaloc::127.0.0.1:19740/Device";
         Object devRef = orb.string_to_object(url);
         device = DeviceHelper.narrow(devRef);

         // Set up one GPRS and one CSD bearer. Both
         // can be set in one method call but for
         // simplicity they are set in separate methods.

         // Set GPRS to slot 0.
         setWirelessBearerGPRS((byte) 0);
         // Set CSD to slot 1.
         setWirelessBearerCSD((byte) 1);

         // Set GPRS as the default connection
         // from slot 0.
         setDefaultConnection((byte) 0);

         // Set up HLA information
         setHLAInformation();

         // Before the Nokia 12 module can receive incoming
         // calls, it must have authentication information set.
         // This has an effect only if at least one
         // CSD bearer is configured.
         setIncomingCallAuthenticationInfo();

         // Set the local IP address used in the PPP layer.
         // Local IP must be set if a PC or mobile phone creates
         // datacall to the Nokia 12 module. Also, remote
         // IMlet loading using the Nokia 12 Configurator
         // requires that an address
         // is set to the target Nokia 12 module.
         setPPPIPAddress();

         log("All parameters set!");

      } catch (Exception e) {
         log(e.toString());
      }

   }

 . . .

}
```

The first step is to create one connection for GPRS to slot 0 and one for CSD to slot 1. The connection that uses GPRS is set as the default connection. This connection is used when an outgoing wireless connection is needed from an

IMlet. The connection that uses CSD is a back-up connection and it can be used to receive incoming data calls.

GPRS bearer

Please note that the Nokia 12 Configurator uses one-based indexing when it refers to locations of stored connections whereas the Wireless Device CORBA API uses zero-based indexing.

Modify the settings according to the running environment. For example, the GPRS access point name 'ap.example.com' is used in the example code and the Configurator UI. Change the name to match the access point name provided by the GSM network operator before running the IMlet.

The timeout is left to 0 (zero) for GPRS (no timeout).

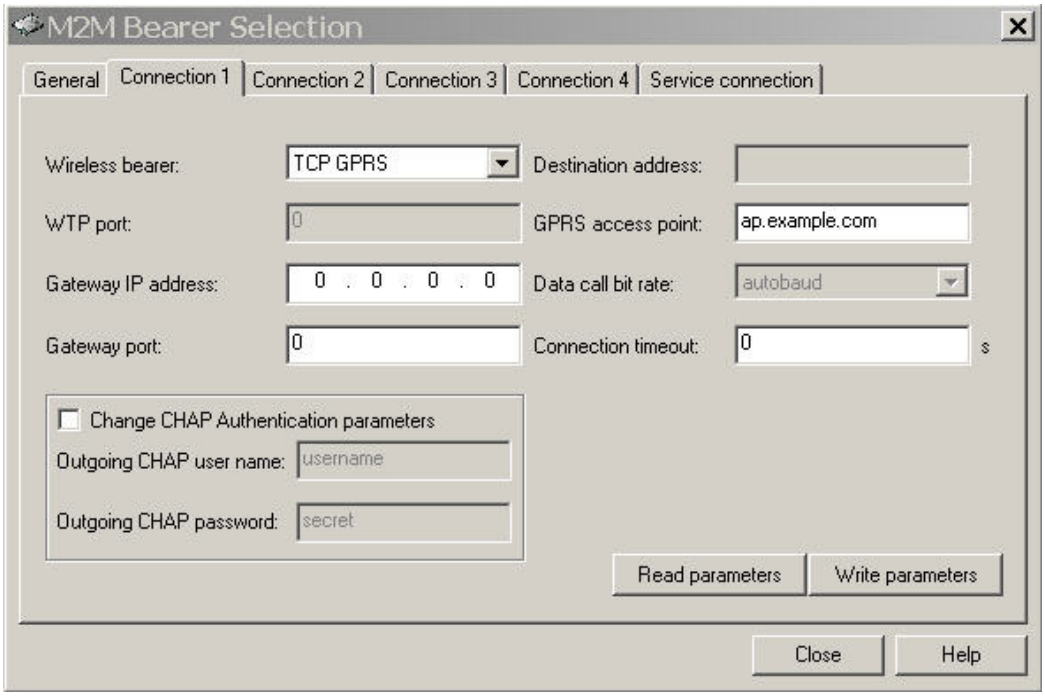

Figure 8. GPRS configuration

The Gateway IP address and Gateway port settings are reserved for WUS and can usually be left empty. WUS allows the server application to initiate GPRS context remotely. See the *Nokia 12 GSM Module Software Developer's Guide* for more information about WUS.

`setWirelessBearerGPRS():`

```
private void setWirelessBearerGPRS(short connectionIndex)
      throws Exception {

      log("Setting wireless bearer GPRS");
```

```
    //
    // Construct the WAPBearerInformation[] structure.
    //

    // Set one wireless bearer. Multiple bearers can be set by one
    // method invocation.
    WAPBearerInformation wapBearerInfoArray[] = new
       WAPBearerInformation[1];

    // Initialise the bearer array.
    for (int i = 0; i < wapBearerInfoArray.length; i++) {
       wapBearerInfoArray[i] = new WAPBearerInformation();
    }

    WAPBearerInformationUnion wapBearerInfoUnionGprs =
       new WAPBearerInformationUnion();

    //
    // Construct the GPRS bearer structure.
    //
    WAPBearerInformationGPRS gprsInfo =
       new WAPBearerInformationGPRS(
          timeoutGprs,
          gwIPAddress,
          gprsAccessPointName,
          pppUserName,
          pppPassword);
    wapBearerInfoUnionGprs.bearerInfoGPRS(gprsInfo);

    wapBearerInfoArray[0].WAPbearerInformationUnion =
       wapBearerInfoUnionGprs;

    // Set port numbers and the connection index
    // zero-based indexing is used here.
    for (int i = 0; i < wapBearerInfoArray.length; i++) {
       // This value is needed only for Nokia 30 compatibility.
       // The value must be set to zero.
       wapBearerInfoArray[i].ETPort = (short) (0);

       wapBearerInfoArray[i].GWPort = gwPort;
       wapBearerInfoArray[i].connectionIndex = connectionIndex;
    }

    WAPParametersHolder wapParameters =
       new WAPParametersHolder(wapBearerInfoArray);

    // Insert the created structure into CORBA Any.
    org.omg.CORBA.Any wapParamAny = orb.create any();
    WAPParametersHelper.insert(wapParamAny, wapBearerInfoArray);

    // Set the parameters.
    device.setParam("WAPParameters", wapParamAny);


    log("Wireless bearer GPRS OK!");

  }
```

CSD bearer

Setting the CSD bearer is very similar to setting the GPRS bearer. A timeout of 120 seconds is used with the CSD bearer to ensure that the data call is closed after 120 seconds if there has been no communications over it during that time.

The Gateway number (modem pool dial number) parameter is left empty intentionally in the Configurator UI and the example code. A valid dial number must be set before the CSD bearer can be used for Mobile-originated (MO) calls. The optional CSD number authentication that is set in the **General** tab is done against this dial number.

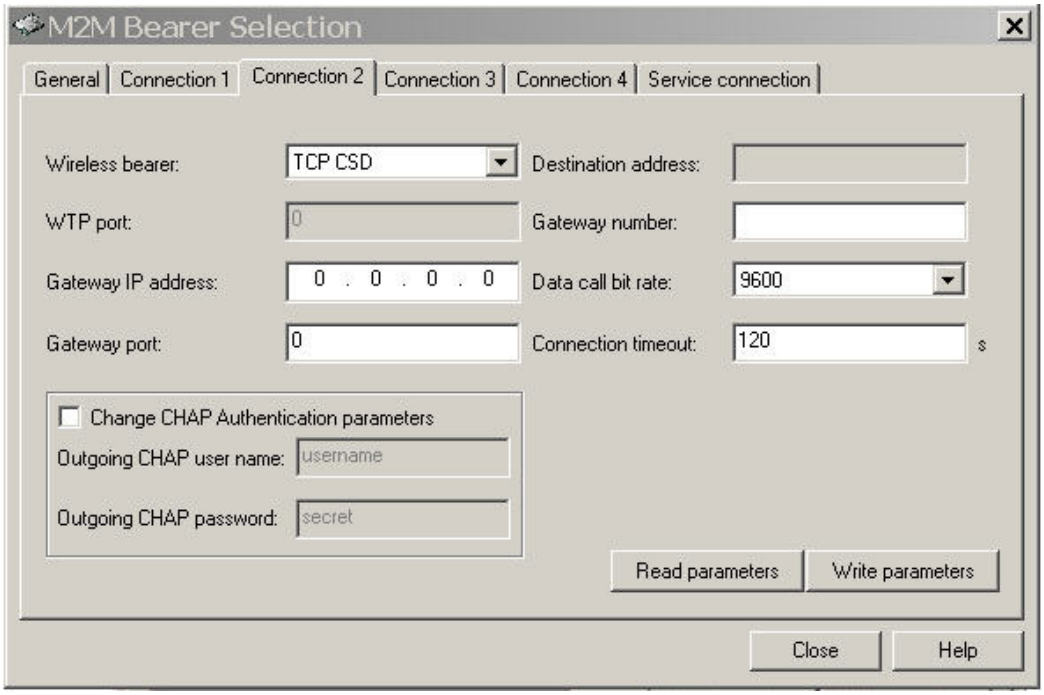

Figure 9. CSD configuration

The Gateway IP address and Gateway port settings are reserved for WUS and can usually be left empty. WUS allows the server application to initiate a CSD data call remotely. See *Nokia 12 GSM Module Software Developer's Guide* for more information about WUS.

`setWirelessBearerCSD()`

```
private void setWirelessBearerCSD(short connectionIndex) throws Exception {

      log("Setting wireless bearer CSD");

      //
      // Construct the WAPBearerInformation[] structure.
      //

      // Set one wireless bearer. Multiple bearers can be set by one
      // method invoke.
      WAPBearerInformation wapBearerInfoArray[] = new
WAPBearerInformation[1];

      // Initialise the bearer array.
      for (int i = 0; i < wapBearerInfoArray.length; i++) {
         wapBearerInfoArray[i] = new WAPBearerInformation();
      }
```

```
      WAPBearerInformationUnion wapBearerInfoUnionCsd =
         new WAPBearerInformationUnion();

      //
      // Create the CSD bearer.
      //

      WAPCSDBitRate bitRate = WAPCSDBitRate.NONTRANSP 9600;

      WAPBearerInformationCSD csdInfo =
         new WAPBearerInformationCSD(
            timeoutCsd,
            bitRate,
            csdModemPoolNumer,
            gwIPAddress,
            pppUserName,
            pppPassword);

      wapBearerInfoUnionCsd.bearerInfoCSD(csdInfo);

      wapBearerInfoArray[0].WAPbearerInformationUnion =
wapBearerInfoUnionCsd;

      // Set the port numbers and the connection index.
      // Zero-based indexing is used here.
      for (int i = 0; i < wapBearerInfoArray.length; i++) {
         // This value is needed only for Nokia 30 compatibility.
         // The value must be set to zero.

         wapBearerInfoArray[i].ETPort = (short) (0);

         wapBearerInfoArray[i].GWPort = gwPort;
         wapBearerInfoArray[i].connectionIndex = connectionIndex;
      }

      WAPParametersHolder wapParameters =
         new WAPParametersHolder(wapBearerInfoArray);

      // Insert the created structure into CORBA Any.
      org.omg.CORBA.Any wapParamAny = orb.create any();
      WAPParametersHelper.insert(wapParamAny, wapBearerInfoArray);

      // Set the parameters.
      device.setParam("WAPParameters", wapParamAny);

      log("Wireless bearer CSD OK!");

   }
```

Other settings

The Nokia 12 Configurator displays the incoming call authentication, Home Location Agent (HLA) information, and default connection settings in the **General** tab. Three methods in the BearerSettings IMlet are used to set this information because all information is not needed in all use cases. For example, the Module ORB and J2ME ORB do not use HLA information in IIOP mode and therefore the HLA settings can be left empty. Also, incoming call authentication information is needed only if one or more CSD connections are set.

Tip: The Nokia 12 module does not use Home Location Agent (HLA) information if ORBs are in the IIOP mode. An application may use the HLA address and HLA

	port fields for example to store server address. Information is stored to permanent memory and can be managed by using `setParam()`, `getParam()` methods from Wireless Device interface. HLA information can be modified also from the Nokia 12 Configurator.

⚠	**Caution:** If one or more CSD connections are set, the incoming call authentication should also be set. If the incoming call authentication is not set, incoming calls are not authenticated. If you know the dial number of the SIM that is set into the Nokia 12 module, you can establish a CSD connection to that Nokia 12 module provided that neither Challenge Handshake Authentication Protocol (CHAP) authentication nor CSD number authentication have been set. If one or more CSD bearers are set into the Nokia 12 module, it will answer to all incoming data calls.

⚠	**Caution:** The Nokia 12 module does not check the PPP username when it is authenticating an incoming call. Only a password check is performed. This means that the caller may use any username, and still establish a connection to the Nokia 12 module as long as it knows the correct password. This feature has been made intentionally to support a large-scale dial-out feature supported by many modem pools. In large-scale dial-out, the modem pool uses its host name as a PPP username when creating calls to remote devices.

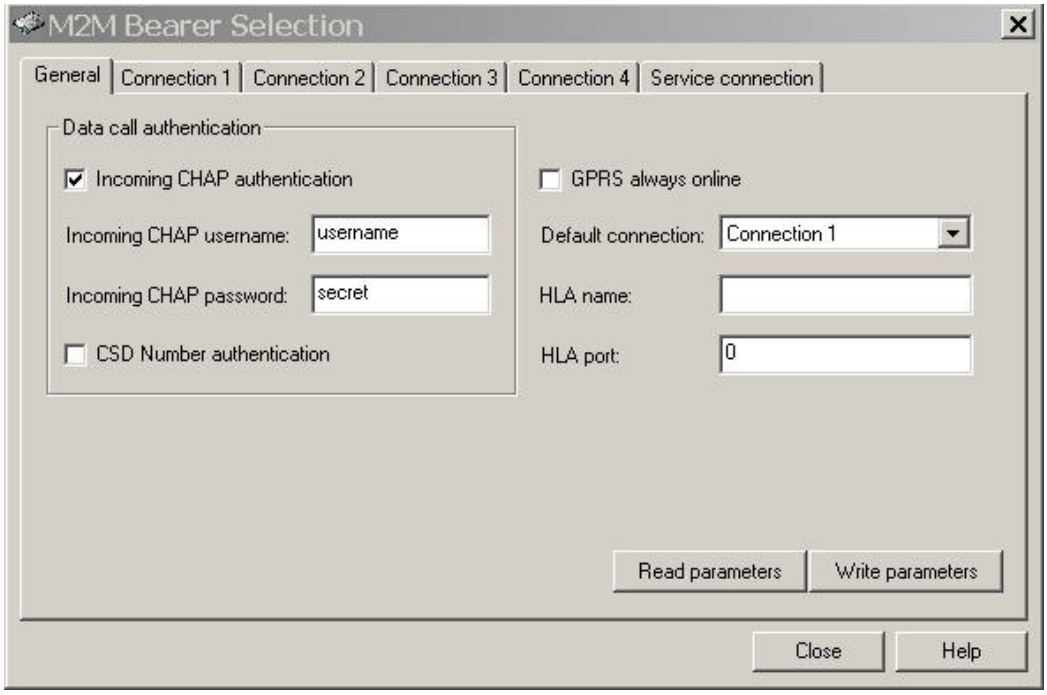

Figure 10. General settings

The connection that uses GPRS and is stored into slot 0 is set as default connection.

`setDefaultConnection()`

```
private void setDefaultConnection(byte connectionIndex) throws Exception {

      log("Setting default connection " + connectionIndex);

      Any any = orb.create_any();
      WAPParametersDefaultConnectionHelper.insert(any, connectionIndex);
      device.setParam("WAPParametersDefaultConnection", any);

      log("Default connection set");

   }
```

The following listing explains the incoming call authentication settings. For security reasons, either the Incoming CHAP authentication or the CSD Number authentication parameter should always be enabled. CHAP authentication is selected for this example.

`setIncomingCallAuthenticationInfo()`

```
private void setIncomingCallAuthenticationInfo() throws Exception {
      log("Setting incoming call authentication information");

   Any any = orb.create any();
      // No CSD number authentication. Enable CHAP authentication
      WAPParametersAuthentication authInfo =
         new WAPParametersAuthentication(
            false,
            true,
            pppUserName,
            pppPassword);

      WAPParametersAuthenticationHelper.insert(any, authInfo);

      device.setParam("WAPParametersAuthentication", any);

      log("Incoming call authentication information OK");

   }
```

HLA information is needed only when the Module ORB or J2ME ORB is set to M2M mode. In M2M mode both the Module ORB and the J2ME ORB need to know the HLA address to be able to publish mobile IORs (MIOR); CORBA object references that can be transported from between devices. The M2M mode is a legacy feature and not recommended to be used in new applications.
`setHLAInformation()`

```
private void setHLAInformation() throws Exception {
   log("Setting HLA information");

   Any any = orb.create any();
   HLAInformation hlaInfo = new HLAInformation(HLAAddress, HLAPort);
   HLAInformationHelper.insert(any, hlaInfo);
   device.setParam("HLAInformation", any);

   log("HLA information OK");
}
```

The local IP address feature is needed when IMlets are remotely updated using the Nokia 12 Configurator. Before the Nokia 12 module can receive an incoming data call from a PC or a mobile phone, its PPP stack must know its own address. Usually the calling device is not able to allocate an IP address to the called Nokia 12 module.

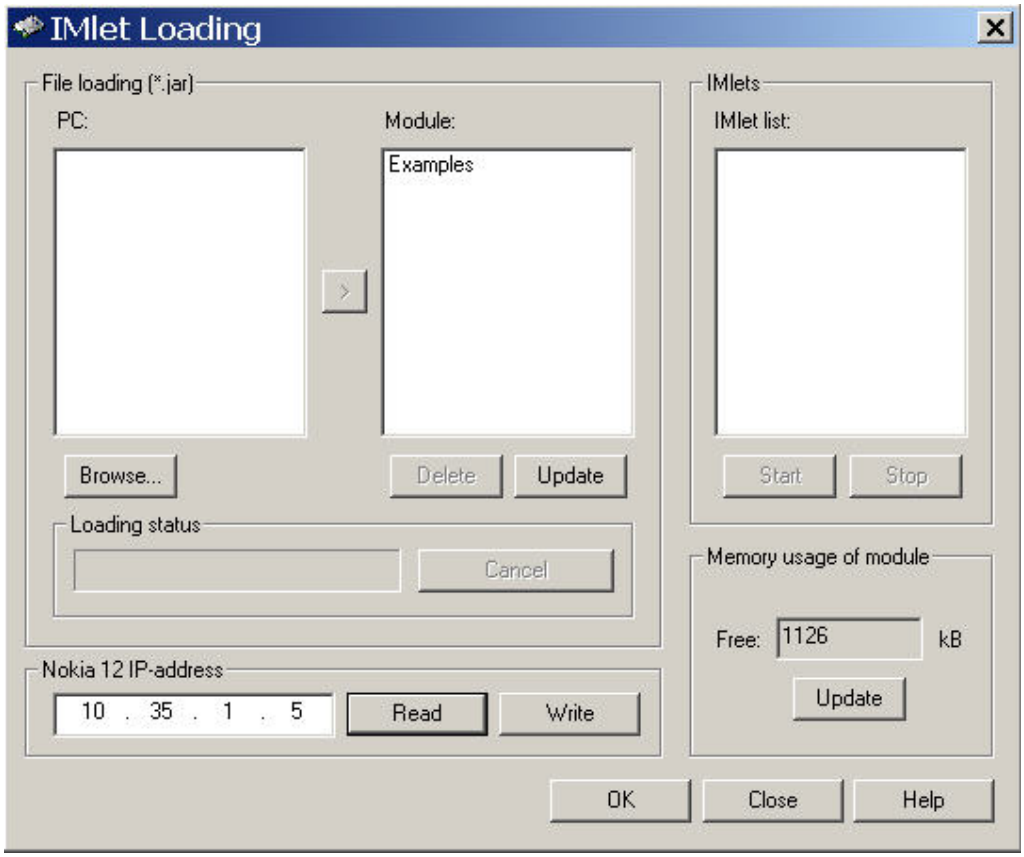

Figure 11. Local PPP IP address

The following code sets the Nokia 12 module IP address from an IMlet.

```
setPPPIPAddress ()
```

```
private void setPPPIPAddress() throws Exception {
   log("Setting PPP IP address");

   Any any = orb.create any();
   PPPNegotiationIPAddressHelper.insert(any, localIpAddress);
   device.setParam("PPPNegotiationIPAddress", any);

   log("PPP IP address OK");

}
```

APPENDIX B: BUILDING TOOLS

AN EXAMPLE OF BUILDING AN IMLET USING ECLIPSE IDE

This chapter guides you how to configure the Eclipse IDE to use the Antenna Ant tasks and how to build IMlets.

Installing the Eclipse

1. Download the Eclipse IDE from www.eclipse.org.
2. Unzip the downloaded zip file to a folder.
3. Start the Eclipse to finish the installation.

Configuring the Antenna

1. Download the Antenna from http://antenna.sourceforge.net.
2. Save the downloaded JAR file to a directory for later use.
3. In Eclipse, choose the **Preferences** command from the **Window** menu. A dialog box presented in Figure 12 opens.

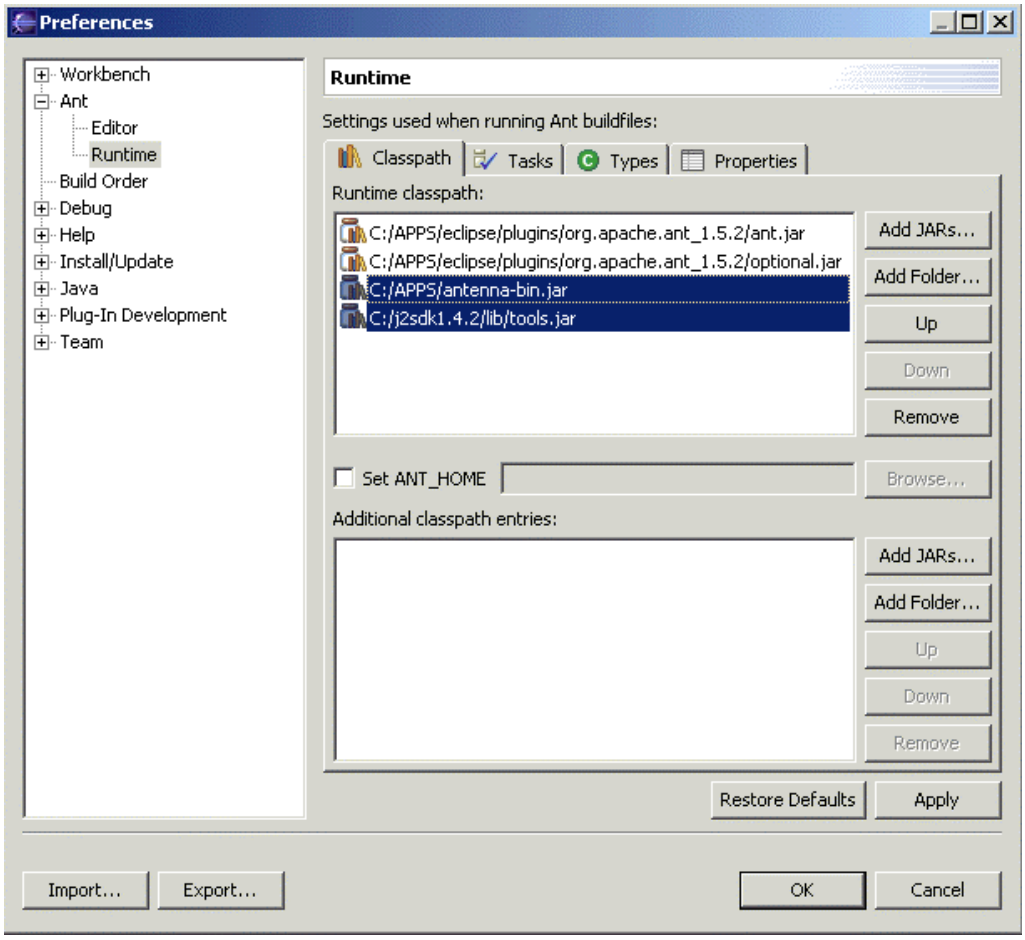

Figure 12. Configuring Eclipse to use Antenna tasks

4. Add the antenna JAR file from the directory where it was saved to the Runtime classpath of the **Classpath** tab by clicking the **Add JARS…** button.

5. Add the **tools.jar** file from the **Java SDK/lib** directory to the Runtime classpath of the **Classpath** tab by clicking the **Add JARS…** button.

Creating an IMlet using Eclipse

1. Create a new project by choosing the **New** ->**Project** command from the **File** menu. A dialog box presented in Figure 13 opens.

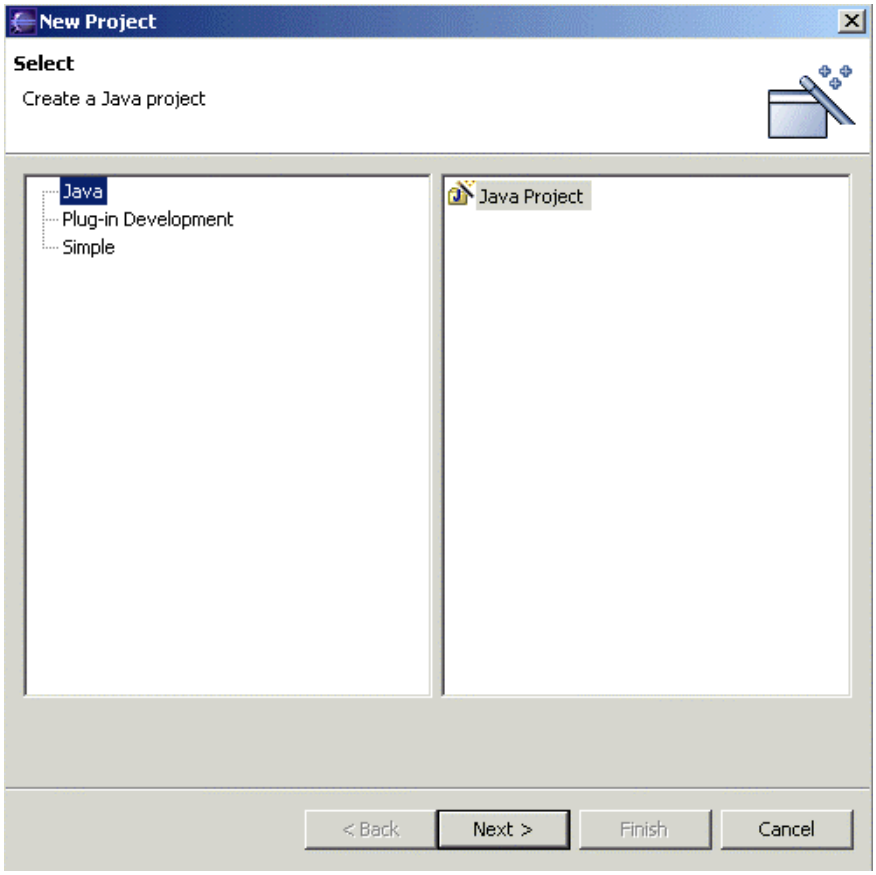

Figure 13. Creating a new project

2. Choose Java from the project list and click **Next**. A dialog box presented in Figure 14 opens.

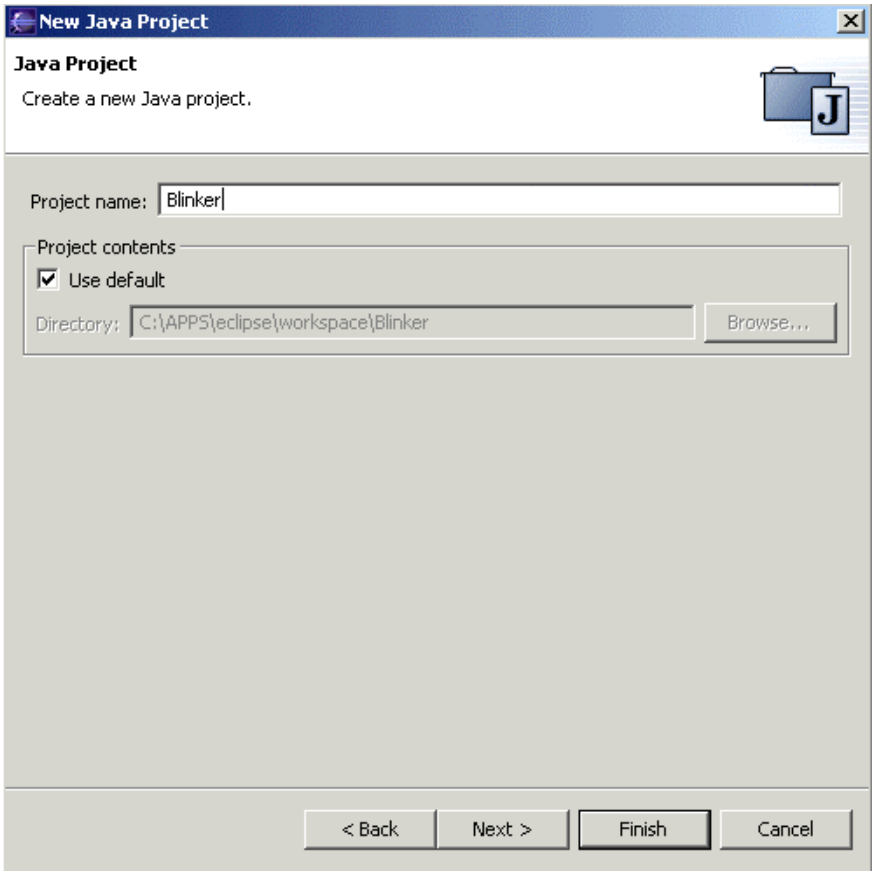

Figure 14. Creating a new Java project

3. Give a name for the Java project. This creates a directory for the project in the workspace directory.

4. Click **Finish**. The Eclipse main window presented in Figure 15 is displayed.

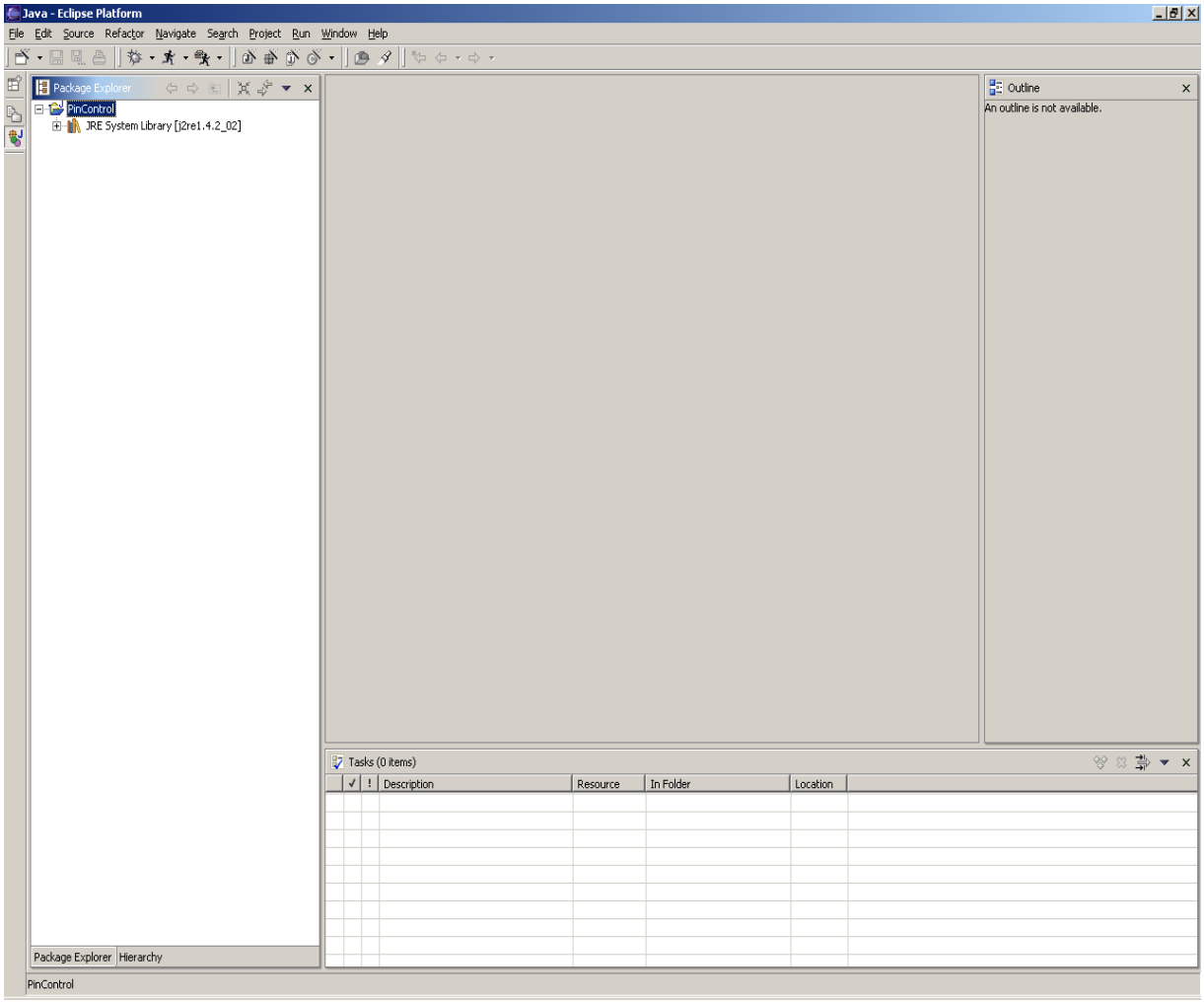

Figure 15. Eclipse IDE before Java build path modification

5. With the right mouse button, select the created Java project and select **Properties** from the drop-down menu. A dialog box presented in Figure 16 opens.

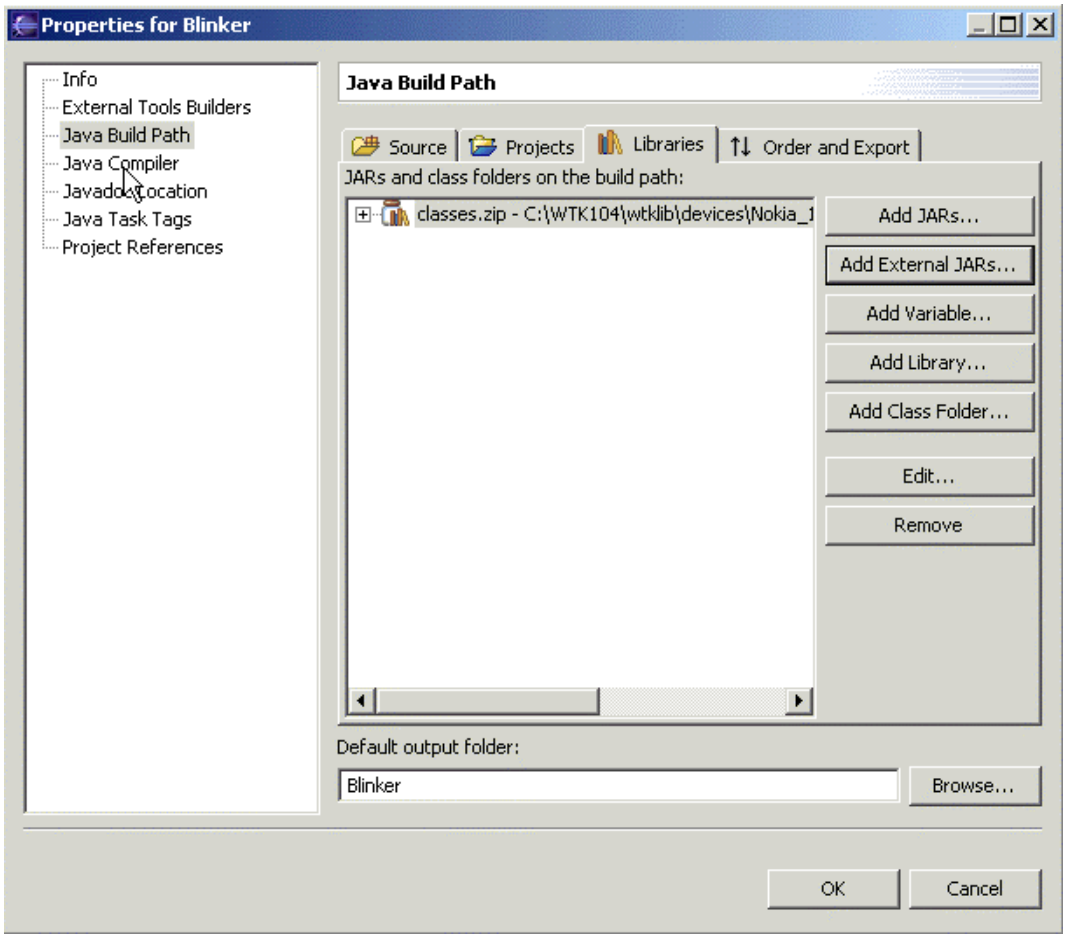

Figure 16. Configuring the build path

6. Select the **Java Build Path** from the left-hand list and then select the **Libraries** tab.

7. Remove the JRE system library by selecting it from the list and pressing the **Remove** button.

8. Add the **classes.zip** file from the Nokia 12 Concept Simulator's **lib** directory to the **Libraries** tab in the **Java Build Path** by pressing the **Add External JARs** button.

9. To return to the Eclipse main window, press the **OK** button.

10. Create a file called **build.xml** into the project directory. Select the project with the right mouse button and select **New** -> **File** from the drop-down menu to create the file. Copy the code of the following **build.xml** file to the created file.

11. To return to the Eclipse main window, press the **Finish** button.

12. Modify the paths according to your system settings as shown below.

- Pointing to Sun WTK 1.0.4 root directory:

```
<property name="wtk.home" value="C:\WTK104"/>
```

- Pointing to Sun java SDK:

```
<property name="java.home" value="C:\j2sdk1.4.2"/>
```

- IMlet name:

```
<property name="app.name" value="Blinker"/>
```

- This is the start class of the IMlet. It must extend the MIDlet:

```
<property name="start.class"
value="com.nokia.m2m.sdk.examples.iocontrol.Blinker"/>
```

- Pointing to the **classes.zip** file from the Nokia 12 Concept simulator installation location:

```
bootclasspath="C:\WTK104\wtklib\devices\Nokia 12 IMP 1 0 Concept Simulator
1 0\lib\classes.zip"
classpath="C:\WTK104\wtklib\devices\Nokia_12_IMP_1_0_Concept_Simulator_1_0\
lib\classes.zip"
```

An example of a **build.xml** file when using Antenna tasks:

```
<?xml version="1.0"?>

<project name="trace" default="build" basedir=".">

    <!-- Define the WTK home directory. Needed by the tasks. -->

    <property name="wtk.home" value="C:\WTK104"/>
    <property name="java.home" value="C:\j2sdk1.4.2"/>
    <property name="app.name" value="Blinker"/>
    <property name="start.class"
value="com.nokia.m2m.sdk.examples.iocontrol.Blinker"/>

    <!-- Define the tasks. -->

    <taskdef name="wtkjad" classname="de.pleumann.antenna.WtkJad"/>
    <taskdef name="wtkbuild" classname="de.pleumann.antenna.WtkBuild"/>
    <taskdef name="wtkpackage" classname="de.pleumann.antenna.WtkPackage"/>
```

```
    <taskdef name="wtkmakeprc" classname="de.pleumann.antenna.WtkMakePrc"/>
    <taskdef name="wtkrun" classname="de.pleumann.antenna.WtkRun"/>
    <taskdef name="wtkpreverify"
classname="de.pleumann.antenna.WtkPreverify"/>
    <taskdef name="wtkobfuscate"
classname="de.pleumann.antenna.WtkObfuscate"/>

    <target  name="build">

        <!-- Create a JAD file. -->

        <wtkjad jadfile="${app.name}.jad"
                jarfile="${app.name}.jar"
                name="${app.name}"
                vendor="John Doe"
                version="1.0.0">

            <midlet name="${app.name}" class="${start.class}"/>
            <!-- List all imlets here -->

            <!--
            <midlet name="Memory" class="Memory"/>
            <midlet name="TestEtObjref" class="TestEtObjref"/>
            -->

        </wtkjad>

        <delete dir="classes"/>
        <mkdir dir="classes"/>

        <!-- Compile everything, but don't preverify (yet). -->

        <wtkbuild srcdir="."
                  destdir="classes"
                  bootclasspath="C:\WTK104\wtklib\devices\
               Nokia_12_IMP_1_0_Concept_Simulator_1_0\lib\classes.zip”
                  preverify="false"/>

        <wtkpackage jarfile="${app.name}.jar"
                    jadfile="${app.name}.jad"
                    classpath="C:\WTK104\wtklib\devices\
               Nokia_12_IMP_1_0_Concept_Simulator_1_0\lib\classes.zip”
                    obfuscate="false"
                    preverify="true">

            <!-- Package our newly compiled classes. -->

            <fileset dir="classes"/>

        </wtkpackage>


      <!-- Start the MIDlet suite -->

      <!-- <wtkrun jadfile="${app.name}.jad" device="
Nokia 12 IMP 1 0 Concept Simulator Beta 0 9" wait="true"/> -->


    </target>

</project>
```

13. With the right mouse button, click your Java project name and select **New -> Class** from the drop-down menu.

14. Enter the package name `com.nokia.m2m.sdk.examples.iocontrol` to the Package field and 'Blinker' to the Name field.

15. To return to the Eclipse main window, press the **Finish** button.

16. Open the created **Blinker.java** file by double clicking it.

17. Copy the code of the Blinker example from Chapter 4.8 to the created **Blinker.java** file in the project.

18. With the right mouse button, click the **build.xml** file in your Java project and select **Run** from the drop-down menu to compile, pre-verify, and package your IMlet. You should be able to do this if the Windows classpath and path are correctly set.

All classes in your Java project are now included in the created IMlet Suite JAR file.

Tip: If the pre-verification fails and an error message 'preverify returned 1' is displayed, it is possible that the **jar.exe** file is not in the system path. The **preverify.exe** file needs the **jar.exe** file from the J2SE SDK to create a JAR file after pre-verification.

USING ANT AND ANTENNA

This chapter describes how to use the Ant build tool from the command line. Before Ant and Antenna can be used, they must be installed. Refer to the Ant and Antenna installation instructions for more information about installation.

The Ant script (**build.xml**) used to build the 'ExampleIMlets' from the Nokia 12 SDK is as follows:

```
<?xml version="1.0"?>

<project name="ExampleIMlets" default="run" basedir=".">

    <!-- Change these according to your system -->
    <property name="bootclasspath"
value="C:\WTK104\wtklib\devices\Nokia 12 IMP 1 0 Concept Simulator 1 0\lib\
classes.zip"/>
    <property name="wtk.home" value="c:\WTK104"/>

    <!-- Define this IMlet suite. -->
    <property name="suite.name" value="ExampleIMlets"/>


    <property name="MicroEdition-Profile_version" value="1.0"/>
    <property name="MicroEdition-Configuration_version" value="1.0"/>


    <!-- Directories of file locations. -->
    <property name="src" value="src"/>
```

```
    <property name="classes" value="classes"/>
    <property name="deploy" value="bin"/>


    <!-- Define the Antenna tasks. -->
    <taskdef name="wtkjad" classname="de.pleumann.antenna.WtkJad"/>
    <taskdef name="wtkbuild" classname="de.pleumann.antenna.WtkBuild"/>
    <taskdef name="wtkpackage" classname="de.pleumann.antenna.WtkPackage"/>
    <taskdef name="wtkmakeprc" classname="de.pleumann.antenna.WtkMakePrc"/>
    <taskdef name="wtkrun" classname="de.pleumann.antenna.WtkRun"/>
    <taskdef name="wtkpreverify"
classname="de.pleumann.antenna.WtkPreverify"/>
    <taskdef name="wtkobfuscate"
classname="de.pleumann.antenna.WtkObfuscate"/>

    <!-- Clean up directories. -->
    <target name="init">

        <echo message="Cleaning up..."/>

        <!-- Delete all output directories -->
        <delete dir="${classes}"/>
        <delete dir="${deploy}"/>

        <!-- Create empty output directories -->
        <mkdir dir="${classes}"/>
        <mkdir dir="${deploy}"/>

    </target>

    <!-- Compile and preverify. -->
    <target name="compile" depends="init">

        <wtkbuild srcdir="${src}"
                  destdir="${classes}"
                  bootclasspath="${bootclasspath}"
                  preverify="true"/>
    </target>


    <!-- Package everything. To obfuscate everything, set
             the corresponding parameter to "true" (requires
             RetroGuard or ProGuard in WTK bin directory or
             in CLASSPATH). -->
    <target name="package" depends="compile">

        <!-- Create a JAD file. -->
        <wtkjad jadfile="${deploy}\${suite.name}.jad"
                jarfile="${deploy}\${suite.name}.jar"
                name="${suite.name}"
                vendor="Nokia"
                version="1.0.0">

           <!-- List IMlets -->
           <midlet name="SerialToTCPBridge"
class="com.nokia.m2m.sdk.examples.socket.SerialToTCPBridge"/>
           <midlet name="Blinker"
class="com.nokia.m2m.sdk.examples.iocontrol.Blinker"/>
           <midlet name="HelloWorldIMlet"
class="com.nokia.m2m.sdk.examples.helloworld.HelloWorldIMlet"/>
           <midlet name="GpsIMlet"
class="com.nokia.m2m.sdk.examples.gps.GpsIMlet"/>
           <midlet name="StopDemo"
class="com.nokia.m2m.sdk.examples.imletmanager.StopDemo"/>
           <midlet name="IOModuleDemo"
class="com.nokia.m2m.sdk.examples.iomodule.IOModuleDemo"/>
           <midlet name="ObjectNames"
class="com.nokia.m2m.sdk.examples.device.ObjectNames"/>
```

```
            <midlet name="DeviceInfo"
class="com.nokia.m2m.sdk.examples.device.DeviceInfo"/>
            <midlet name="SignalQualityIMlet"
class="com.nokia.m2m.sdk.examples.device.SignalQualityIMlet"/>
            <midlet name="BearerSettings"
class="com.nokia.m2m.sdk.examples.device.BearerSettings"/>
            <midlet name="ETIMlet"
class="com.nokia.m2m.sdk.examples.terminal.ETIMlet"/>
            <midlet name="HTTPIMlet"
class="com.nokia.m2m.sdk.examples.http.HTTPIMlet"/>
            <midlet name="HTTPParametersIMlet"
class="com.nokia.m2m.sdk.examples.http.HTTPParametersIMlet"/>
            <midlet name="WatchdogIMlet"
class="com.nokia.m2m.sdk.examples.watchdog.WatchdogIMlet"/>
            <midlet name="TCPClient"
class="com.nokia.m2m.sdk.examples.socket.TCPClient"/>
            <midlet name="TCPServer"
class="com.nokia.m2m.sdk.examples.socket.TCPServer"/>
            <midlet name="DatagramClient"
class="com.nokia.m2m.sdk.examples.socket.DatagramClient"/>
            <midlet name="DatagramServer"
class="com.nokia.m2m.sdk.examples.socket.DatagramServer"/>
            <midlet name="HTTPServerIMlet"
class="com.nokia.m2m.sdk.examples.socket.HTTPServerIMlet"/>
            <midlet name="HelloClientIMlet"
class="com.nokia.m2m.sdk.examples.hello.impl.HelloClientImlet"/>
            <midlet name="HelloServerIMlet"
class="com.nokia.m2m.sdk.examples.hello.impl.HelloServerImlet"/>

            <midlet name="IOObserverIMlet"
class="com.nokia.m2m.sdk.examples.iomodule.IOObserverIMlet"/>
            <midlet name="MessageObserverIMlet"
class="com.nokia.m2m.sdk.examples.device.MessageObserverIMlet"/>
            <midlet name="PortModeIMlet"
class="com.nokia.m2m.sdk.examples.device.PortModeIMlet"/>

        </wtkjad>

        <!-- Create a JAR file and update the JAD file. -->
        <wtkpackage jarfile="${deploy}\${suite.name}.jar"
                    jadfile="${deploy}\${suite.name}.jad"
                    classpath="${bootclasspath}"
                    obfuscate="false"
                    preverify="true"
                    profile="IMP-1.0">

            <!-- Package our newly compiled classes. -->
            <fileset dir="${classes}"/>

        </wtkpackage>

    </target>


    <target name="run" depends="package">

        <!-- Start the MIDlet suite -->
        <wtkrun jadfile="${deploy}\${suite.name}.jad"
device="Nokia_12_IMP_1_0_Concept_Simulator_1_0" wait="true"/>

    </target>

</project>
```

Modify the paths located at the beginning of the script to match the paths in your system.

The following command runs the Ant. It executes the **build.xml** file from the directory where the command is run. The Ant must be in the system path.

```
ant
```

This command builds sources from the **src** directory and places the generated JAR and JAD files into the **bin** directory.

APPENDIX C: AN EXAMPLE OF CONFIGURING THE NOKIA 12 GSM MODULE FROM IMLET

Sometimes it is necessary to change the configuration parameters of the Nokia 12 module from IMlet code. An IMlet may require that the Nokia 12 module has certain settings before it can execute correctly. You can also set these settings using the Nokia 12 Configurator, but it requires manual configuration work for each installed Nokia 12 module.

An IMlet needs to execute the following tasks to modify the Nokia 12 module configuration parameters:

1. Check the current configuration using the `getParam()` method from the Wireless Device interface.
2. Change configuration values, if needed.
3. Reset the Nokia 12 module if configuration changes were made. The Nokia 12 module starts to use a new configuration after the reset.

This example application is designed so that it requires the following serial port configurations:

- serial port 1 is in GPS mode,
- serial port 2 is in SystemProtocol mode for the Nokia 12 Configurator and
- serial port 3 is reserved for Java.

The IMlet checks the configuration at each startup and changes it if the configuration is not correct. An example IMlet code for changing the serial port mode is located in *com.nokia.m2m.sdk.examples.device.ModuleUtils* class of the Nokia 12 SDK.
All configuration parameters are set and get using the `device.setParameter()` and `device.getParameter()` CORBA interfaces.

The *com.nokia.m2m.orb.idl.wirelessdevice.Device* class has an operation for parameter reading:

```
void getParam(String parameterName, AnyHolder
    responseData) throws ParamFailureException
```

The available parameter names are described in the *Nokia 12 GSM Module Properties Reference Guide*.

As the *Nokia 12 GSM Module Properties Reference Guide* defines, the `RS232PortModeList` parameter is used to access RS232 port settings. An IMlet can ask for port information using following code:

```
Object deviceRef =
 orb.string_to_object("corbaloc::127.0.0.1:19740/Device");
com.nokia.m2m.orb.idl.wirelessdevice.Device device =
               DeviceHelper.narrow(deviceRef);

AnyHolder anyHolder = new AnyHolder();
device.getParam("RS232PortModeList", anyHolder);
```

Because the parameter name was `RS232PortModeList`, there is a `RS232PortModeListHelper` class in the Java API. Use the helper class to extract data from the returned CORBA `Any` object.

```
RS232ModeInformation[] currentModes =
   RS232PortModeListHelper.extract(anyHolder.value);

// List current modes to log
log("Portmodes: ");
for (int i = 0; i < currentModes.length; i++) {
   log("port "
         + currentModes[i].port
         + " mode "
         + currentModes[i].mode.value()
         + "\r\n");
}
```

The `RS232PortModeListHelper` extracts an array of `RS232ModeInformation` objects.

The next step is to check whether the port mode has to be changed.

```
// Check each port against target setup
boolean changes = false;
for (int i = 0; i < currentModes.length; i++) {

   // Check only port 1 and port 3 because only the settings
   // of those ports can be modified
   if (currentModes[i].port == 1
                  && currentModes[i].mode != RS232Mode.GPS) {
      changes = true;
   } else if (currentModes[i].port == 3
                  && currentModes[i].mode != RS232Mode.JAVA) {
      changes = true;
   }
}
```

Change the parameters if the setup was not correct.

```
if (changes) {
   log("Change settings");
```

```
   RS232ModeInformation port1Mode = new RS232ModeInformation(1,
      RS232Mode.GPS);
   RS232ModeInformation port3Mode = new RS232ModeInformation(3,
      RS232Mode.JAVA);

   RS232ModeInformation[] newInformation = new RS232ModeInformation[] {
      port1Mode, port3Mode };

   Any any = orb.create any();
   RS232PortModeListHelper.insert(any, newInformation);
   device.setParam("RS232PortModeList", any);

   device.reset();
   // Module reboots after 10 seconds
   return true;
} else {
   log("Serial port configuration is OK");
}
```

The last step is to reset the Nokia 12 module (if setup changes were made). The IMlet checks the configuration again after the restart, and finds the configuration to be correct.

Tip: If you accidentally create an IMlet, which resets the Nokia 12 module immediately after each start-up, just let it reset 50 times in row. The Nokia 12 module will clear the IMlet start-up record after 50 sequential resets, and does not attempt to run the IMlet any more. See Chapter 3.5.1 for more information.